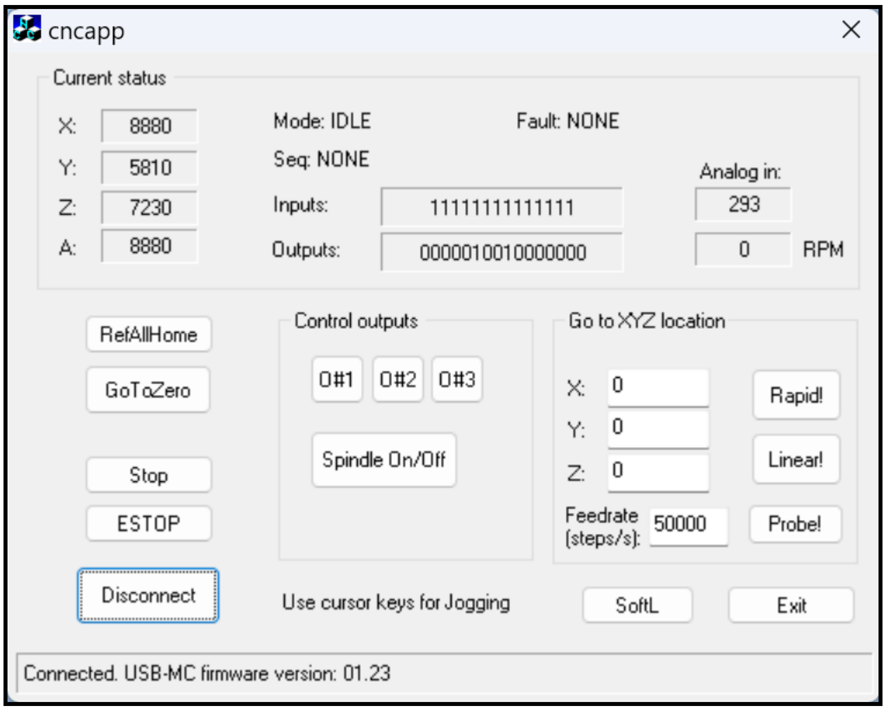
Programming Library

For Audioms Automatika doo USB motion controllers



Connected. USB-MC firmware version: 01.23

AUDIOMS AUTOMATIKA

www.audiohms.com

TABLE OF CONTENTS

USB Motion Controllers Programming Library

Introduction

Audioms Automatika doo USB motion controllers use USB for connection with computer. WinUSB is used as underlying base but programmer does not need to know anything about USB standard since all low level communication is handled by the library itself and hidden from the user.

Just as information for curious ones, multiple endpoints are used for simultaneous full duplex communicating lines over USB bus. Bulk transfer is used for motion buffer data and high priority interrupt transfer mode is used for other commands to the controller.

Supported motion controllers

USB programming library supports following line of Audioms Automatika doo motion controllers:

- ISO-USB-BOX motion controller,
- USB-MC-INT motion controller and
- USB-MC motion controller.

Using the library

To use USB-MC library, application should include header file **usbmclib.h** and also add **usbmc.lib** to the project for linking. Header file contains all functions and structures declarations used by the library. Dynamic link library file **usbmc.dll** contains library code and should be placed in the folder where executable for the application is located.

Opening and initializing device

When USB-MC motion controller is powered on by connecting it to the computer it will enter safe mode where all outputs are in hiZ (high impedance) mode. Status LED blinks slowly.

Application should use **usbmc_Open()** function to open connection to USB-MC motion controller.
Next, to initialize controller, function **usbmc_Init()** should be called. This function is used to set almost all configuration parameters for the controller and i/o pin configuration.
After successful initialization USB-MC controller will enter on-line mode. This is normal operating mode where digital outputs are enabled, as well as all other functions.
Status LED is continuously lit.

While in on-line mode, USB-MC controller sends status packet to the computer every 50ms. This concept serves as a "watch dog" thus making sure that connection is present and that both sides (controller and software) are alive and well. Application should periodically check if a status packet is available by calling **usbmc_GetStatus()**. This can be done in a timer callback or in a thread loop that serves for all other maintenance.

Application concept example

After initializing the USB motion controller application should do continual monitoring of the controller status (axis positions, input states etc.) and do its processing, motion generation etc.
One way to organize application is to use timer function for continuous background processing. For example, Windows API function **SetTimer()** can be used to initialize timer callback function that will be called every 25 ms to do its processing:

```
tid = SetTimer(hwnd, 0, 25, (TIMERPROC)UpdateProc);

VOID CALLBACK UpdateProc( HWND hwnd, UINT uMsg, UINT_PTR idEvent, DWORD
dwTime)
{
        if(online && usbmc_GetStatus(&mcstat)){        //if there is a new status packet
                                                       //available

                //examine new status packet and do appropriate processing

        }

        //Do other jobs like motion generation or setting output states as needed

        if(online && update_out ){
                //Set all outputs
                usbmc_CmdSetOutputs(dig_outputs, mc_flags, zero_mask);
                zero_mask = 0;
                update_out = 0;
        }
}
```

Using windows timer is the easy way, but it comes with some caveats. If the computer is at heavy load some timer messages (and timer callbacks) may not be issued which can lead to skipping and stuttering. Also, as timer callback is executed in the same thread as UI, UI drawing and user operations may delay critical events from execution.

For most reliable application a multi-threaded concept should be used. This may not be too much different from described example at first glance. Instead of **SetTime**r(), windows function **CreateWaitableTimer()** can be used to create timer object that will periodically set an event. Work thread loop would wait for the event using **WaitForSingleObject()** and when event gets signaled, it can do its status check and processing.

Off course, any multi-threaded application requires careful synchronization between threads.

USB programming library is thread safe in a way that it will not allow two API function calls that call to hardware via USB to be executed at the same time. This calls will be synchronized so that while first thread is executing one API function, other threads will wait.

Working modes (states)

Status packet has member field **state** that indicates current state of the controller. It may return one of the following values.

```
enum Mode{
        M_IDLE=0,   //connected, on-line and no operation in progress
        M_JOG,      //jogging one or more axis
        M_SEGM,     //g-code buffered motion in progress
        M_HOLD,     //g-code buffer has received data but motion not yet started
        M_ESTOP,    //ESTOP signaled by the ESTOP input, limit switches etc.
        M_SAFE,     //controller is in safe, HiZ outputs mode
        M_RDOWN     //after probe hit there is a ramp down for all axes
};
```

The Audioms Automatika doo USB motion controllers are designed so that it has two main motion generation mechanisms that are mutually exclusive. These are:

- Jog mode and
- Buffered motion generation modes.

Jogging is initiated by application using appropriate API functions **usbmc_CmdJogOn() / usbmc_JogOff()** and then is completely performed by hardware. Controller generates acceleration ramp in real-time according to the previously setup parameters and generates Step&Dir signals. So, jog mode is immediate and fast thus it is mainly used for manual control of the machine. In jog mode every axis is independent and no coordinated motion of multiple axes is possible.

On the other hand, for moving along some arbitrary path we need to coordinate motion of two or more axes. This is where buffered motion mode is used. In this case the application is generating position data for all axes and sends it to controller for execution. Motion planning task and generating axis positions in time may be complex job but it is perfectly well suited to the computer.

USB-MC controller expects position data for all axes in 1ms time intervals. When generating Step&Dir signals from this data USB-MC motion controller uses interpolation to connect this 1ms time points (segments) into smooth trajectory.
So application prepares data, and this data is sent to the controller one buffer at a time. USB-MC controller has internal buffer memory for 1000 6-coordinate points (1 second of motion practically). While working in this mode, application can check current motion buffer state and decide when to send more data so that controller maintains continuous motion generation without starving out of data.
In practice, application does not really need to do this controller hardware buffer checks and buffer sending since it is performed by the library.

Buffered motion programming

Application can use **usbmc_BufAdd()** function to add 6-coordinate points to the motion buffer. Before adding more data it should check if there is a free space in the buffer by calling **usbmc_BufFreeCnt()**. This motion data is added to the buffer maintained by the

library and to transfer this data to the USB-MC controller, function **usbmc_BufUpdate()** should be called regularly. This library function internally checks buffer state of the controller and library buffer and performs buffered data transfers when needed.
As soon as USB-MC controller receives any motion data it will enter M_HOLD mode, waiting for more data. After the controller receives enough data (500ms), it enters M_SEG mode and starts to execute buffer data and to generate Step&Dir signals.
As long as new data is added to the buffer and buffer update is called regularly, uninterrupted smooth motion will continue.

In case that motion generated is short and it ends in less than 500ms, library function **usbmc_CmdPurge()** can be called to release for execution all data in the controller buffer. After all data is consumed controller will revert to M_IDLE mode.

Example addition to the timer loop:

```
        AppMotionGeneration();    //Generate motion data and call usbmc_BufAdd()

        usbmc_BufUpdate();        //Regularly send current data in the buffer to the
controller

        //If move was short and finished before enough data is generated to auto-trigger
        //buffer execution start, then purge all that is currently left in the buffer

        if(mcstat.state==M_HOLD && !GMOVE_BUSY){
                usbmc_CmdPurge(0);
        }
```

Sequences

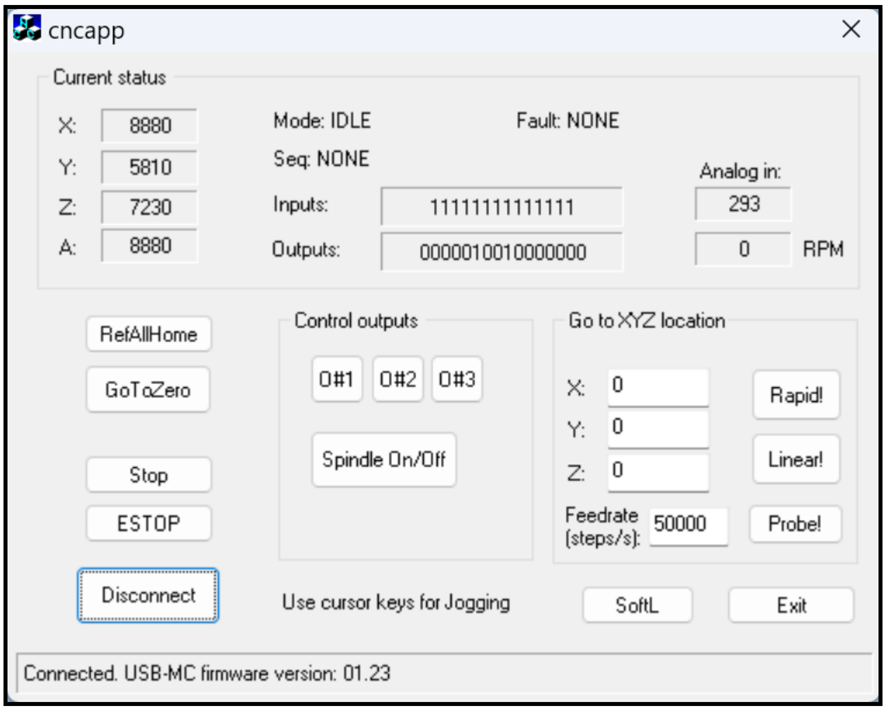
USB-MC supports sequences that may consist of multiple moves like homing and probing. Sequences are performed pretty much autonomously by the controller and/or library and application can monitor current progress by examining status packet.

Homing for any axis is initiated by function usbmc_CmdHomeAxis() Multiple axis can be homed at the same time.
Probing is performed by function usbmc_CmdProbe() function.

Application must make sure that no sequence or other operation is currently active before attempting to initiate any new sequence.

Example CNC application

Source code for the functional example CNC application that is based on this library is provided for demonstration (Figure 1). The application uses most of the library functions including opening and initializing motion controller, jogging, buffered motion generation, homing, probing etc.



X: 8880
Y: 5810
Z: 7230
A: 8880
Mode: IDLE
Fault: NONE
Seq: NONE
Inputs: 11111111111111
Outputs: 0000010010000000
Analog in: 293
0 RPM
X: 0
Y: 0
Z: 0
Feedrate (steps/s): 50000

Connected. USB-MC firmware version: 01.23

Figure 1 Example CNC application

C++ source code is provided as Microsoft Visual Studio 2008 project. Any later version of Visual Studio should be able to open and compile the project. The application is based on MFC framework, so MFC should be installed in the Visual Studio.

USB-MC API functions reference

Hardware communication functions

BOOL usbmc_Open();

Opens USB device. Must be called before calling any other API function. Only one application can open controller at one time.

Returns TRUE if motion controller is connected and opened successfully.
In case of an error returns FALSE.

---

void usbmc_Close();

Close USB device when finished using it.

BOOL usbmc_Init(USBMC_Config *ucfg, USBMC_IOCfg *pcfg);

Initialization of the motion controller parameters. Must be called after opening the device to configure various parameters and enter on-line (normal) working mode.
After the device is initialized, it enters on-line mode where digital outputs are enabled and status packet is sent every 50 ms to the application.

Parameters:
*ucfg, pointer to USBMC_Config structure, to config various controller parameters
*pcfg, pointer to USBMC_IOCfg structure, to config input and output pins

If initialization is successful function returns TRUE. Otherwise it returns FALSE.

BOOL usbmc_GetStatus(USBMC_Status *stat);

When new status packet is available this function copies status structure to the supplied pointer and returns TRUE.
If new status is not available at this time function returns FALSE.

USBMC_Status *stat - Status structure contains current axis positions, states for all digital inputs, analog input etc.

Application must periodically check if new status packet is available by calling this function. Status packet is sent every 50ms by the controller and ensuring that packets are read in a timely manner ensures that there is a reliable connection.
If application stalls too much reading for status packets the motion controller would perceive this as a problem on the computer side and enter safe hiZ mode for outputs, also entering off-line mode.
Application should also do its checking so if no status packet is available for some time (for example 500ms) application should interpret this as a problem on the motion controller side and signal error to the user, close connection or activate ESTOP mode.

In either case, to re-establish on-line normal working mode, function usbmc_Init() should be called.

---

void usbmc_Disconnect();

End on-line mode of the controller. Outputs are placed in hiZ mode, status packets are not sent any more.
To re-establish on-line normal working mode, function usbmc_Init should be called.

BOOL usbmc_CmdStop(BYTE flags=0);

Calling this function will stop all active operations (jog, g-code segment move, home, probe sequences etc.). Motion buffer is cleared so any data that is present in the buffer will be lost.

In addition to this, one or more of the following flags can be used:

```
enum StopModes{
        STOP_ESTOP = 1,                 //activate ESTOP mode
        STOP_DISCONNECT = 2             //terminate on-line mode
};
```

Returns TRUE if operation was successful.

BOOL usbmc_CmdSetOutputs(WORD out_bits, WORD flags=0, WORD zmask=0);

Set states for all digital outputs. Also additional functions are available, it is done in this way because it is more efficient to do it all in one call since data sent is very small in size.

WORD out_bits – 16-bit bit mask for out states, bit0= out1, bit1= out2 etc.

WORD flags – bitwise OR of one or more of the following:

```
enum RuntimeFlags{
        SOFT_LIMITS    = 1,     //activate soft limits
        SHUTTLE_MODE = 2,       //shuttle mode is not currently supported, but
                                //using this flag will activate hardware feedhold
};
```

WORD zmask – bit mask for axes that should be set to zero. Bit0 – X axis, bit1 – Y axis, etc. Axis position is set to zero for example when referencing is asked by user but there is no home switch configured for the axis.

Returns TRUE if operation was successful.

BOOL usbmc_CmdJogOn(int axis, BOOL dir, DWORD speed);

Start Jogging the axis. Controller generates trapezoidal acceleration profile based on the preconfigured parameters and generates STEP and DIR pulses for selected axis. Axis will continue jogging until usbmc_CmdJogOff() function is called or Stop or Estop function is activated. All axes are independent, that is jog for any axis can be started and stopped any time.

If the axis is already in jogging motion, new speed is applied. The axis speed will change to this new value using acceleration that is configured for that axis.

Important note: application must make sure that all other operations are finished and that controller is in M_IDLE mode or M_JOG mode (check status packet) before attempting to enter jog mode. it is not permitted to start jog mode while for example controller is in M_SEGM mode where motion data is still present in the buffer.

Parameters:

int axis – 0 based index of axis to jog, 0=X, 1=Y, etc.
BOOL dir – jog direction, 0=negative direction, 1=positive direction
DWORD speed – jog speed in Steps/Sec

Returns TRUE if operation was successful.

BOOL usbmc_CmdJogOff(int axis);

Stop jogging the specified axis. Axis will decelerate to stop.

int axis – 0 based index of axis to stop, 0=X, 1=Y, etc.

Returns TRUE if operation was successful.

BOOL usbmc_CmdSpindle(BOOL on, WORD pwm_duty=0, DWORD speed=0);

This command will set spindle state as on or off and also it is used to set **pwm_duty** (used for PWM spindle motor control) and **speed** (used for Step&Dir spindle motor control).
This command is not used to set spindle relay(s). Relays (if needed) are programmed by the application using any available digital output(s).

Parameters:
BOOL on – 1=spindle on, 0=spindle off,
WORD pwm_duty – 16-bit duty cycle (0-65535) represents duty cycle 0-100% (used only in PWM spindle mode)
DWORD speed – spindle speed in steps/sec (used only in Step&Dir spindle mode)

Returns TRUE if operation was successful.

BOOL usbmc_CmdHomeAxis(BYTE axis);

Start referencing sequence for the given axis. Axis will start to move toward the home switch using configured speed. When the switch is activated axis will move in the opposite direction until switch is deactivated.
Simulations referencing of all axes is possible.

Important: before starting the homing sequence application should make sure that no other operation or sequence is in progress and that controller is in M_IDLE state.

Application can check status packet for homing operation progress. Sequence information will indicate homing stage.

When one axis is finished homing the status packet member field **homed** will indicate that in this way:

```
BOOL axis_homed = (homed & S_FLG_AXISHOMED);
int axnum = (homed & 7);  //number of axis
```

Returns TRUE if operation was successful.

BOOL usbmc_CmdProbe(double start[6], double end[6], double feedrate);

Start probing sequence move. Linear move with trapezoidal acceleration is generated by the library and sent to the controller. Controller will continuously check Probe input while moving and as soon as Probe input is activated it will store current axes locations and decelerate to stop.

Application can periodically call **usbmc_GetProbeResult()** to query status of the probing sequence. When this function returns PROBE_HIT, application can use **usbmc_GetProbePos()** to get hit position for all axes.

Parameters:
double start[6] – x,y,z... coordinates for start point in steps,
double end[6] – x,y,z... coordinates for end point in steps,
double feedrate – feedrate in mm/s
Returns TRUE if operation was successful.

BOOL usbmc_GetProbePos(double hit_pos[6]);

Call this function to get probe hit position after successful probe move. If there was no successful probing result this function returns FALSE.

Parameters:
double hit_pos[6] – x,y,z... coordinates on hit for all axes

Returns TRUE if operation was successful.

---

int usbmc_GetProbeResult();

This function is used to check probing status.
It can return one of the following values as defined in ProbeState enum:

PROBE_IDLE – no probing in progress
PROBE_ACTIVE – probing is active
PROBE_NOHIT – probing move finished with NO hit
PROBE_HIT – probing move finished with successful hit

BOOL usbmc_IsProbing();

Can be used to check if probing operation is in progress.

BOOL usbmc_IsHoming();

Can be used to check if any axis is currently homing.

---

void usbmc_PosResync();

Application maintains current position for axes for buffered motion that it generates but after the motion functions that controller performs autonomously (like Jogging) or after abrupt STOP where current position is lost and unknown, software must resync to the current position that controller achieved.

Whether there is a need for resync is indicated by the bit in status packet member (**flags & S_FLG_SWRESYNC**). After all axes finish jogging, controller will indicate that position resync should be performed by setting this bit.

Calling this function will set current position that is maintained by the library to the one that is indicated by the status fields **axis_ticks[]**
Application should do the same, that is store current values in **axis_ticks[]** as a current position (or calculate from steps to mm units for example) so that any subsequent buffered motion computation has correct starting point.

BOOL usbmc_SetupMPG(BYTE axis, double step, DWORD maxvel, BOOL enable);

Setup and enable/disable hardware MPG.

Parameters:
BYTE axis – 0 based index for axis that is controlled by MPG,
double step – move step for one encoder count,
DWORD maxvel – maximum axis velocity when controlled by MPG,
BOOL enable – 1=enable or 0=disable MPG control

Returns TRUE if operation was successful.

---

const char *usbmc_GetFirmwareVersion();

Function returns pointer to the string that contains USB-MC controller firmware version.

Motion buffer management functions

BOOL usbmc_BufAdd(USBMC_BufSeg *p);

Add one segment (point) to the motion buffer. Application should first check if there is a free space in the buffer by calling **usbmc_BufFreeCnt().**
Application is adding points to the buffer maintained by the library and not to the controller directly. Library has motion buffer for max 4000 points. Application should only periodically call **usbmc_BufUpdate()** that will do all the work for transferring data from the library buffer via USB to the motion controller.

USBMC_BufSeg *p – pointer to the structure that has coordinates for 6 axes, also may have additional data and commands that should be synchronized with motion stream.

Returns TRUE if operation was successful.

---

int usbmc_BufFreeCnt();

Function returns free space in the motion buffer. Library maintains motion buffer that has capacity of 4000 points. Application does not have to use full buffer capacity, when adding the data it may choose to use for example max 1000 points in the buffer in this way:

```
if(usbmc_BufFreeCnt()>3000)     //if there is more than 3000 free points in the buff
        usbmc_BufAdd(&p);       //add one more point
```

---

void usbmc_BufClear();

Clear all data in the library motion buffer.

BOOL usbmc_BufEmpty();

Check if any data is present in the motion buffer that is maintained by the library. Return value is TRUE if buffer is empty, FALSE otherwise.

BOOL usbmc_CmdPurge(BOOL clear=FALSE);

Execute all data left in the USB-MC controller hardware buffer if any (or clear all data).

BOOL clear – if clear==FALSE, if there is any data in the USB-MC buffer, it is released for execution
if clear==TRUE, any data that is present in the USB-MC buffer is cleared and will not be executed.

Returns TRUE if operation was successful.

---

int usbmc_BufUpdate(int max_cnt=75);

This function should be called regularly so that library can maintain buffer transfers to the USB-MC motion controller as needed.

int max_cnt – maximum points to send at one time. As this function is called often (for example every 25ms) the quantity of data sent per call does not have to be excessive.

---

void usbmc_SetBuffSize(int moves);

Can be used to set target maximum USB-MC buffer occupation that library will try to achieve. Value smaller than maximum and default (1000) can be used so that effectively as smaller amount of data is in the buffer the latency is also smaller. Too small value may lead to unreliable connection and discontinuities in the motion.

int moves – maximum count of moves (6 coordinate points) that will be placed in the controller buffer. Default and maximum value is 1000.

USB-MC structures

struct USBMC_AxisCfg

```
//Configuration parameters for one axis

typedef struct {
        //Motor tuning
        int spu;        //steps per unit
        int vel;        //velocity steps/s
        int acc;        //acceleration steps/s2

        BYTE slave;  //master axis (for A,B or C)

        //soft limits
        int smin, smax;        //min and max coordinate in steps

        //homing
        int home_speed;      //in steps/s
        int home_offset;
        BYTE  home_flags;    //see enum HomeFlags

}USBMC_AxisCfg;
```

struct USBMC_SpindleConfig

```
typedef struct{
        BYTE spindle_motor;
        int spindle_pwmfreq;
        int spindle_pwmmin;
        int spindle_pwmmax;
} USBMC_SpindleConfig;
```

struct USBMC_Config

```
//All configuration parameters for USB-MC controller
typedef struct {
        USBMC_AxisCfg ax_cfg[7];         //7 axes configs (6 axes + spindle)
        USBMC_SpindleConfig sp_cfg;   //Spindle config

        BYTE state_flags;   //see enum CfgFlags

        int home_retract_speed;    // steps/s
        BYTE home_deref_speed;          ////
```

```
	BYTE enc_for_mpg;		//encoder that is used for MPG (0 or 1)

} USBMC_Config;
```

---

struct USBMC_BufSeg

```
// One motion buffer segment, 1ms update rate

typedef struct {
	double pos[6];		//axis position in steps
	DWORD line;			//this segment ID number (optional)
	BYTE ecmd:4, epin:4;	//fast ext out command and pin number (optional)
	BYTE pwm_duty;		//8-bit pwm duty for compensation (optional)
} USBMC_BufSeg;
```

---

struct USBMC_Signal

```
typedef struct {
	BYTE pin;	//pin number >=1 (0=undefined)
	BYTE enable:1, inv:1;	//signal enable and invert (make active low)
} USBMC_Signal;
```

---

struct USBMC_IOCfg

```
//Configuration parameters for input and output pins

typedef struct{

	//Output signals
	USBMC_Signal stepx, stepy, stepz, stepa, stepb, stepc, steps;	//Step outputs
	USBMC_Signal dirx, diry, dirz, dira, dirb, dirc, dirs;	//Dir outputs
	USBMC_Signal charge1, charge2;		//Charge pump signal outputs
	USBMC_Signal ext1, ext2, ext3, ext4, ext5, ext6;	//External out 1-6

	//Input signals
	USBMC_Signal homex, homey, homez, homea, homeb, homec;	//home switches
	USBMC_Signal limitxp, limitxn, limityp, limityn, limitzp, limitzn;	//limit switches
positive and negative
	USBMC_Signal limitap, limitan, limitbp, limitbn, limitcp, limitcn;
	USBMC_Signal index;	//Spindle index input
	USBMC_Signal estop;	//ESTOP input
	USBMC_Signal enc1a, enc1b;	//Encoder1 A,B inputs
	USBMC_Signal enc2a, enc2b;	//Encoder2 A,B inputs
	USBMC_Signal probe;
```

```
        WORD debounce[16];      //debounce data for all input pins,
debounce[0]=debounce value for pin1 etc.

}USBMC_IOCfg;
```

---

struct USBMC_Status

```
//Motion controller complete status report
typedef struct {
        BYTE size;              //size of this struct
        BYTE state;             //current mode, see enum Mode values
        BYTE cur_seq;           //current sequence, see enum Sequence values
        BYTE flags;             //various flags, see enum StatusFlags values
        WORD inputs;            //input states
        WORD outputs;           //output states
        int axis_ticks[7];      //all axes positions in steps
        float spindle_rpm;      //rpm detected using Index input
        WORD inbuff;            //points currently in hardware buffer
        DWORD time;             //timestamp when this status was created
        int encoder1_pos;       //encoder1 position
        int encoder2_pos;       //encoder2 position
        BYTE fault;             //fault code, see enum Fault values
        BYTE homed;             //signals referenced axis
        DWORD gline;            //id from cur buffer segment (like prog line currently executing)
        WORD analogv;           //analog input value
} USBMC_Status;
```

USB-MC enumerations

```
        //Current mode
        enum Mode{
                M_IDLE=0,
                M_JOG,
                M_SEGM,
                M_HOLD,
                M_ESTOP,
                M_SAFE,
                M_RDOWN
        };
```

```
        //Sequences
        enum Sequence{
                SQ_NONE=0,
                SQ_HOME0,
		SQ_HOME1,
		SQ_HOME2,
		SQ_HOME3,
		SQ_HOME4,
		SQ_PROBE1,
		SQ_PROBE2,
		SQ_MPG
	};
```

```
	//Stop reasons
	enum Fault{
		F_NONE=0,
		F_UNK,
		F_HOMESW,
		F_LIMITSW,
		F_SOFTLIMIT,
		F_ESTOP,
		F_PROBE
	};

	enum HomeFlags{
		// Home flags
		HOME_NEG = 1,
		HOME_AUTOZERO =	2,
	};

	enum SpindleTypes{
		//Spindle types
		SPINDLE_NOMOTOR	= 0,
		SPINDLE_PWM,
		SPINDLE_STEPDIR
	};

	enum EngineCmd{
		//Engine commands
		C_EXT_OUT_ON	=	1,
		C_EXT_OUT_OFF =	2
	};

	enum StopModes{
		STOP_ESTOP = 1,
		STOP_DISCONNECT	= 2
	};

	//cfg state FLAGS
	enum CfgFlags{
		//CHARGE flags
		CHARGE_5K =	1,
		CHARGE_ALWAYSON	=	2,
		//Misc
		LIMIT_OVERRIDE =	4,
		//THC_MODE	=	8,
		LASER_COMP =	16,
		LASER_GRAY =	32,
		//SOFTLIMITS_SLOWZONE =		64,		//unused
		HOME_UNLINK_SLAVE =		128	//=!HOME_SLAVEWMASTER
		LOW_SPEED = 256,	//use 125kHz mode instead of full 250kHz
					//Step signal width is twice larger (4uS)
	};

	//Runtime state flags
	enum RuntimeFlags{
		SOFT_LIMITS	= 1,
		SHUTTLE_MODE = 2,
		// OFFLINE_MODE=4
		// THC_MODE	=	8
		// THC_DISABLEUPDN	=	32
		// PROBE_LIMIT	=	64
	};

	//Status flags
	enum StatusFlags{
		S_FLG_SWRESYNC		=	1,
		S_FLG_THCON	=	2,
		S_FLG_SPINDLEON		=	 4,
		S_FLG_THCARCOK		=	8,
		S_FLG_THCUP	=	16,
		S_FLG_THCDOWN		=	32,
		S_FLG_THCLOCK =	64,

		S_FLG_AXISHOMED = 128,
	};

	enum ProbeState{
		PROBE_IDLE=0,
		PROBE_ACTIVE,
		PROBE_NOHIT,
		PROBE_HIT
	};
```

DOCUMENT REVISION:

- Ver. 1.0, August 2024, Initial version