

## Implementing $\mu$ Vision2 DLL's for Interface to Hardware Debuggers

**Application Note 145 Rev. 2**

---

May 28, 2000, Munich, Germany

Revision History:      Feb 24, 2000 Initial Version  
                             May 28, 2000 changed Address Representation for 8x51

by      Peter Holzer, Keil Elektronik GmbH    support.intl@keil.com    ++49 89 456040-0

The  $\mu$ Vision2 Debugger supports direct interface to hardware debuggers like monitors or emulators. This interface is done via an **Advanced Generic Debugger Interface** called AGDI. The AGDI interface is independent of the controller architecture is flexible, easy to implement and introduces only minimal overhead. It performs the interface to all basic debugger features, allows complex breakpoints and can be expanded with emulator or target specific commands, dialogs or display pages that appear in the same way as  $\mu$ Vision2 dialogs.

To ease the development of a target driver, the AGDI interface and configuration framework is provided in the SampTarg project. SampTarg, is a synonym for 'Sample Target Driver'. It is a ready to run driver with remote setup and all of the AGDI functions provided as dummies. The driver consists of a Visual-C++ (6.0) project file and the following source files:

SampTarg.cpp,h:	main file, AppWizard created. Provides target setup and startup code
SetupT.cpp,h:	code sample for a setup dialog
AGDI.H:	prototypes for the AGDI functions (rarely modified)
BOM.H:	various prototypes and definitions (do not modify !)
ComTyp.H:	some type and other definitions (do not modify !)
Collect.h:	local driver definitions, for use by target driver programmer.
TestDlg.cpp,h:	the model for a modeless extension dialog.

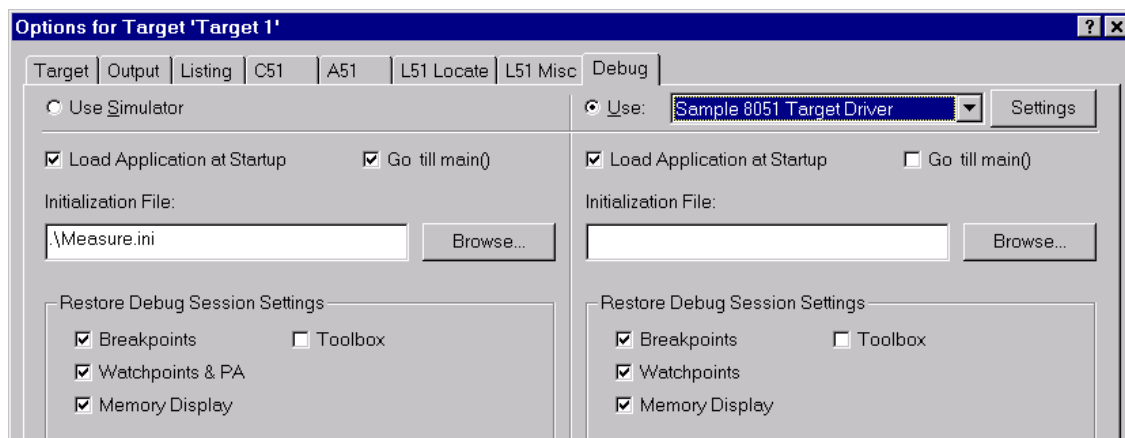
In order to develop a target driver, knowledge about C/C++ programming and the MS Visual-C++ 6.00 Programming Environment is required.

## How to use the Sample Target Driver

In order to use the Sample target driver, you must perform the following steps:

- Install  $\mu$ Vision2 and the C51 Compiler on your machine.
- Create a folder such as D:\Src32\Target\
- Unzip the file **SampTarg.zip** into the folder. Make sure that the 'use folder names' checkbox is checked since SampTarg uses some subfolders.

- Add the following line to the file 'TOOLS.INI', section 'C51':  
`TDRV0=D:\Src32\Target\Debug\SampTarg.DLL ("Sample 8051 Target Driver")`  
 Note: if TDRV0 is already in use, then use the next free digit, for example TDRV1.
- Start Visual-C, select the 'SampTarg.dsw' project file.
- Select 'Build – Set active configuration', choose the SampTarg Win32 Debug configuration.
- Select 'Build – Rebuild All' to create the driver.
- Select 'Project – Settings'. Click at the 'Debug' tab. Browse for the 'Executable for Debug session'. You need to select the file Uv2.Exe. It is normally in C:\Keil\Bin but this depends on where you have installed  $\mu$ Vision2. After that, close the dialog.
- Run  $\mu$ Vision2 by pressing the F5 key. Select 'Project – Open Project', the Select Project dialog comes up. Select the 'Measure.uv2' project. It can be found normally in the folder - **C:\Keil\C51\ExamplesMeasure**. Select 'Rebuild all target files' to build the project.
- Select 'Options for Target – Debug'. From the combobox, select "Sample 8051 Target Driver" which is our sample driver. Make sure that the 'Use:' radio button is checked. If everything is right, then the dialog should look like this:



- Close the dialog.
- Select 'Debug – Start/Stop Debug Session'. This will start the  $\mu$ Vision2 debugger. It initializes and loads our SampTarg.DLL. Click the file 'Measure.c' in the  $\mu$ Vision2's Files tab. Scroll to the 'main()'. You will notice the light and dark gray area to the left of the window, the dark gray ranges identify lines with executable code. Open the Disassembly window (View – Disassembly Window). It shows the disassembled instructions of the Measure application.

Open the Memory window: from the menu, select 'View – Memory Window'. Enter 'C:0x0000' followed by <Enter> in the address entry field of the memory window. You should see the code memory bytes starting at code address 0x0000.

Note that the Sample driver contains code for the memory interface, the register interface and breakpoint management. The code for Go and Step provides just dummies, you can't therefore start execution of a user program.

## Implementing the Driver: Required Steps

In order to connect the SampTarg driver to your hardware, you should perform the following steps:

- Setup your target hardware
- Write code to implement the connection between the target driver and your hardware. You can use the serial port or any other resource. Add the file(s) to the SampTarg project.
- Connect your driver code to the appropriate functions contained in AGDI.CPP. Search for the following comment in AGDI.CPP:

```
//--- Interface functions between AGDI and target follow
//-----
```

The comment is followed by a set of functions such as ReadData(). These functions need to be connected to your communications code. The first set of functions which need to be completed are InitTarget(), ReInitTarget() and StopTarget(). They are required for proper startup and shutdown of the driver. Then the memory access functions, register access functions should be connected.

- The final set of functions required are Step(), GoCmd() and SetClrBp(). These deal with execution of the user program and breakpoint setup and clear. Note that the GoCmd() should not return until the execution is stopped, either by reaching a code breakpoint or some other event.
- Test your driver. If the basic functions are running, switch SampTarg into Release Mode and rebuild it. If this is complete, test the driver again. Note that the 'Release' folder has the release version of the driver. This requires you to change the TOOLS.INI file:  
`TDRV0=D:\Src32\Target\Release\SampTarg.DLL ("Sample 8051 Target Driver")`  
 Watch out for general protection faults. They happen very likely if for example null or invalid pointers are passed fourth and back.

## Example: SampTarg-51 Interface

The DLL Driver Name for external Display DLL's is stored in the file C:\KEIL\TOOLS.INI. An new driver is installed by adding the name and path to the C:\KEIL\TOOLS.INI file. Each CPU family has it's own section in the TOOLS.INI file.

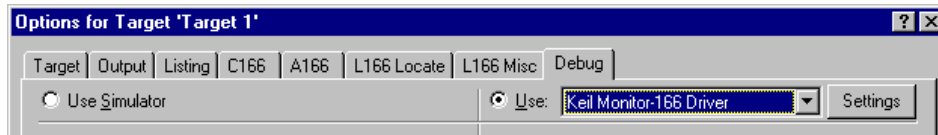
### Example for a TOOLS.INI file:

```
[UV2]
ORGANIZATION="Keil Elektronik GmbH"
:

[C51]
PATH="C:\Keil\C51"
BOOK0=HLP\RELEASE.TXT("Release Notes")
:
TDRV0=SampTarg\SampTarg.DLL ("Sample 8051 Target Driver")
:
```

You need to modify the TOOLS.INI file. For example, if you want to add the SampTarg driver to the list of target drivers, add the line reading TDRV0=xxx in the [C51] section. If you don't have TRDV entries, the start with TDRV0=xxx ( $\mu$ Vision2 accepts up to 40 drivers, TDRV0...TDRV39).

The line TDRV0=SampTarg\SampTarg.DLL ("Sample 8051 Target Driver") specifies the SampTarg.DLL in the SampTarg folder which is in C:\Keil\C51 (as specified by PATH). You can change this folder to suit your needs. The embraced string ("Sample 8051 Target Driver") is used by  $\mu$ Vision2 to show the driver in the list of drivers in the Options for Target sheet:



*Note: the combo shows 'Keil Monitor-166 Driver', it should read 'Sample 8051 Target Driver'*

The display DLL's that should be used during system debugging can be selected in the Options for Target - Debug page under the Use drop down menu. With the Settings button you may implement a configuration dialog for the debugger interface. Configuration data may be stored directly in the  $\mu$ Vision2 project file, the registry or a separate INI file. The configuration data in the project file allows you project specify settings of your debugger. At the time the  $\mu$ Vision2 debugger is started, the debugger DLL is automatically loaded and initialized.

## Target Driver System Remote Setup

Once the Target driver is selected, pressing 'Settings' in the Options for Target Sheet causes  $\mu$ Vision2 to load the driver DLL. After that,  $\mu$ Vision2 calls the function **DllUv3Cap** 2 times with the following function codes:

```
int _EXPO_ DllUv3Cap (DWORD nCode, void *p) ;  
    nCode 2:      // first call: match family  
    nCode 1:      // second call: Cpu/Target-DLL Settings
```

For nCode 2, the return value must be 8051 (0x1F73) in case of an 8051 target or 80167 (0x13927) in case of a 80166/80167 target. The following sample code shows how the remote setup is performed. The QDLL structure has been extracted out of 'Bom.h' to get an easy overview of the data layout. Note that at the time DllUv3Cap() is called,  $\mu$ Vision2 is not in debugging mode. Therefore, you can't access any debug resources.

For details, refer to the SampTarg.cpp file. This source file contains the sample code for remote setup and command options parsing. Also provided there is an entry to run a setup dialog. The source code for the setup dialog is in SetupT.cpp and SetupT.h. The dialog is used to specify a com port number and the baudrate. You may change this dialog to fit your needs.

Note that at the time of remote setup, you can't access any debug resources of  $\mu$ Vision2, because  $\mu$ Vision2 is not in debugging mode. This implies also that none of the AGDI functions are accessed by  $\mu$ Vision2.

## AGDI Interface Functions

All functions that start with **AG\_** need to be defined in the target driver DLL. If a function executes correctly, the value 0 is returned to  $\mu$ Vision2, otherwise an error code should be returned. Note that the following exported functions must be defined, either fully functional or just as dummies:

**AG\_Init, AG\_MemAtt, AG\_BpInfo, AG\_BreakFunc, AG\_GoStep, AG\_Serial, AG\_MemAcc, AG\_RegAcc, AG\_AllReg, AG\_HistFunc**

If this is not the case,  $\mu$ Vision2 considers the target driver as invalid and cancels using it.

## Start Debugging

When the Debugger is started with Debug - Start/Stop Debug Session an enum function is called. Depending on the target architecture, the name of the function is as follows:

---

<b>80166/80167:</b>	<b>int EnumUv3167 (void *p, DWORD nCode)</b>
<b>8051:</b>	<b>int EnumUv351 (void *p, DWORD nCode)</b>
<b>80251:</b>	<b>int EnumUv3251 (void *p, DWORD nCode)</b>

μVision2 calls this function first to match the cpu family of the current project with the target driver family. Then this function is called again with nCode = 2 and the pointer **p** being a pointer to **struct dbgblk** (for more information refer to COMTYP.H).

The code can be found in file SampTarg.cpp.

If the EnumUv3xxx function completes successfully, then μVision2 starts initializing the AGDI interface by calling the **AG\_Init** several times, each time initializing or requesting a different item.

The code for all of **AG\_** functions can be found in file AGDI.CPP.

The first call contains the function code nCode = **AG\_INITFEATURES** and requires the target driver to set the support features:

```
supp.MemAccR = 0;      // memory-access while execting
supp.RegAccR = 0;      // register-access while exectuing
supp.hTrace = 0;       // trace support
supp.hCover = 0;       // code coverage support
supp.hPaLyze = 0;      // Performance-Analyzer support
supp.hMemMap = 0;      // Memory-Map support
```

The shown values in the sample code are typical for monitors, where resources such as registers or memory can't be accessed while a user program is executed. This of course may be different if your hardware supports such features.

You may consult the SampTarg.cpp file which provides a good starting point.

The next calls to **AG\_Init** with nCode = **AG\_INITITEM** are repeated several times with sub-codes. These serve the purpose of transferring handles, pointers and other stuff to your driver.

The next series of **AG\_Init** calls have nCode = **AG\_GETFEATURES** with the sub-codes to query the actual features:

```
case AG_GETFEATURE:      // extract some feature
    switch (nCode & 0x00FF) {
        case AG_F_MEMACCR:    nE = supp.MemAccR; break;
        case AG_F_REGACCR:    nE = supp.RegAccR; break;
        case AG_F_TRACE:      nE = supp.hTrace; break;
        case AG_F_COVERAGE:   nE = supp.hCover; break;
        case AG_F_PALYZE:      nE = supp.hPaLyze; break;
        case AG_F_MEMMAP:      nE = supp.hMemMap; break;
    }
    break;
```

## System Reset

When the μVision2 - **Reset** Toolbar button is given, **AG\_Init** nCode = **AG\_EXECITEM** | **AG\_RESET** is called which then in turn calls ResetTarget(), which is an empty function. You should put whatever code is required in there to reset your target system. μVision2 assumes that after this **AG\_INIT** call the system is in reset state.

## Stop Debugging

When the debugging session stops, uVision2 activates **AG\_Init** with nCode = **AG\_EXECITEM** | **AG\_UNINIT**. You should close the connection to your target and free any allocated dynamic memory.

---

## The CPU Register Interface

The **Project Window - Regs** tab layout is fully defined in the target driver.  $\mu$ Vision2 expects that the Target Driver uses the callback *pCbFunc* (*AG\_CB\_INITREGV*, &*dsc*). when *AG\_Init* with *nCode* = *AG\_GETFEATURE* is called. The call to *pCbFunc*(*AG\_CB\_INITREGV*, &*dsc*); can be given several times and at any time to change the layout of the **Project Window - Regs** tab dynamically.

The *struct REGDSC* defines the register layout (refer to the file AGDI.H for more information).

```
struct RegDsc {
    I32      nGitems;        // number of group items
    I32      nRitems;        // number of register items
    RGROUP   *GrpArr;        // array of group descriptors
    RITEM     *RegArr;        // array of register descriptors
    void      (*RegGet) (RITEM *vp, int nR); // get RegItem's value
    I32      (*RegSet) (RITEM *vp, GVAL *pV); // set RegItem's value
};
```

The File AGDI.CPP contains code for the 8051 register layout. The first item is the group definition:

```
struct rGroup rGroups[] = {
    { 0x00, 0x01, "Regs", }, // Group 0, show expanded
    { 0x00, 0x01, "Sys", },  // Group 0, show expanded
};
```

The group names are shown in the register view, these items act as parent for the child items. The next definition represents the register items:

```
struct rItem rItems[] = {
//--desc-nGi-nItem-szReg[--isPC-cc-iHig--
    { 0x01, 0, 0x00, "r0", 0, 1, 0, },
    { 0x01, 0, 0x01, "r1", 0, 1, 0, },
    { 0x01, 0, 0x02, "r2", 0, 1, 0, },
    { 0x01, 0, 0x03, "r3", 0, 1, 0, },
    { 0x01, 0, 0x04, "r4", 0, 1, 0, },
    { 0x01, 0, 0x05, "r5", 0, 1, 0, },
    { 0x01, 0, 0x06, "r6", 0, 1, 0, },
    { 0x01, 0, 0x07, "r7", 0, 1, 0, },
};
```

The register items shown are assigned to group 0, which is the 'Regs' group. The 'desc' member must be initialized to 1. The 'nItem' member is a number assigned by you, it should be in range 0x00...0xFF. The 'szReg' member represents the name of register, the 'isPC' member must be set if the register represents the program counter only. The 'cc' item means 'can be changed'. It should be set to 0 to avoid value changes via the register view. The 'iHigh' member is used to force the register view to draw the name and value highlighted.

```
{ 0x01, 1, 0x10, "a", 0, 1, 0, },
{ 0x01, 1, 0x11, "b", 0, 1, 0, },
{ 0x01, 1, 0x12, "sp", 0, 1, 0, },
{ 0x01, 1, 0x13, "dptr", 0, 1, 0, },
{ 0x01, 1, 0x14, "PC $", 1, 1, 0, },
{ 0x01, 1, 0x100, "psw", 0, 1, 0, },
```

The register items shown before are assigned to group 1, the 'Sys' group. Note that the 'nItem' numbers need to be different.

A special case is represented by item 'psw', it uses the item number 0x100 (any number which is a multiple of 0x100 such as 0x200, 0x300, ... is ok too). In this case,  $\mu$ Vision2 treats the item as a parent of children. Such a parent can be collapsed and expanded. The requirement for the children is that their numbers must be in range 'parent-number + 1 ... parent-number + 0xFF'. The sample code uses the numbers 0x101 to 0x107 for representing the flag bits of the psw:

```
{ 0x01, 1, 0x101, "p", 0, 1, 0, }, // child-0 of 'psw' item
{ 0x01, 1, 0x102, "f1", 0, 1, 0, }, // child-1 of 'psw' item
```

```
{ 0x01, 1, 0x103,"ov",    0, 1, 0, }, // ...
{ 0x01, 1, 0x104,"rs",    0, 1, 0, }, // ...
{ 0x01, 1, 0x105,"f0",    0, 1, 0, },
{ 0x01, 1, 0x106,"ac",    0, 1, 0, },
{ 0x01, 1, 0x107,"cy",    0, 1, 0, },
```

Note that you can use many items that behave like ‘psw’, as long as each item gets a multiple of 0x100 for it’s item number.

When  $\mu$ Vision2 needs to draw the register view, it accesses the RegGet() function to obtain the current value of a register. At this point, you must fetch the register value out of your target to the descriptor. You should study the functions RegGet and RegSet() in AGDI.CPP.

It should be noted that  $\mu$ Vision2 gives an announcement to RegGet() before fetching the individual register items:

```
switch (nR & 0xF0000000) {
    case UPR_NORMAL:    // Setup Normal Regs
                        // read all registers out of your target here
// . . .
```

The announcement gives you the chance to read all the registers in one junk, repetitive accesses to the target can be avoided. Once again, take a look at the implementation of RegGet/RegSet in AGDI.CPP.

The register interface is registered in  $\mu$ Vision2 by setting up a REGDSC structure and by using the AG\_CB\_INITREGV callback as shown in the InitRegs() function:

```
static void InitRegs (void) {
    REGDSC    dsc;

    dsc.nGItems = sizeof (rGroups) / sizeof (rGroups[0]); // number of group-items
    dsc.nRItems = sizeof (rItems) / sizeof (rItems[0]);   // number of reg-items
    dsc.GrpArr  = rGroups;                                // &rGroups[0]
    dsc.RegArr  = rItems;                                  // &rItems[0]
    dsc.RegGet  = RegGet;                                  // register get function
    dsc.RegSet  = RegSet;                                  // register set function
    pCbFunc (AG_CB_INITREGV, &dsc);                       // Install RegView in uVision2
}
```

If you want to change the register view layout dynamically, simply setup a REGDSC structure with another set of rItems and rGroups before using the AG\_CB\_INITREGV callback again.

$\mu$ Vision2 uses the AG\_AllReg() function to fetch or store the registers as defined by the RG51 typedef in AGDI.H:

```
U32 _EXPO AG_AllReg (U16 nCode, void *vp); // load or store all register values.
```

The AG\_RegAcc() function is used to access individual register item values. This happens most likely when the name of a register is part of an expression in the command line of  $\mu$ Vision2, for example:

‘DPTR = 0x1200 | R7’:

```
U32 _EXPO AG_RegAcc (U16 nCode, U32 nReg, GVAL *pV); // load/store individual register value
```

## Address Representation

The representation of the addresses depends on the architecture. When using the 80166/80167 architecture,  $\mu$ Vision2 converts all addresses regardless of their type such as near, far, huge or otherwise DPPn based into a linear value. This means that for example the address 0x20000 needs to access the physical memory address 0x20000 of your target.

In the 8051 world, things are different: addresses are actually 8, 16 or 24 bit offsets, depending on the memory space to access. The AGDI interface as well as  $\mu$ Vision2 use a 32-bit address value, where the

---

lower 8 or 16 bit represent the offset into the given memory space, and the most significant byte of the address represents a memory space selector value, as defined in AGDI.H:

```
#define amNONE 0x0000 // not spaced
#define amIDATA 0x00F3 // IDATA
#define amDATA 0x00F0 // DATA
#define amXDATA 0x0001 // XDATA
#define amBIT 0x00F1 // BIT
#define amPDATA 0x00FE // PDATA (maps to XDATA)
#define amCODE 0x00FF // CODE
```

The code address C:0x1234 is represented as  $(\text{amCODE} \ll 24) | 0x1234$  which yields 0xFF001234. The data address D:0x72 is represented as  $(\text{amDATA} \ll 24) | 0x0072$  which yields 0xF0000072. The external data address X:0x5678 is represented as  $(\text{amXDATA} \ll 24) | 0x5678$  which yields 0x01005678. If a target supports more than 64K of xdata, then the address uses a three byte offset where the third byte represents the segment such as  $(\text{amXDATA} \ll 24) | 0x12AAEE$ . This applies also for targets which support more than 64K of code:  $(\text{amCODE} \ll 24) | 0x035555$  is an example where the code address 0x5555 in segment 0x03 is specified. The advantage is that the memory space selector, the segment and the offset are combined into a single value.

Take a look at the ReadMem() and WriteMem() functions in AGDI.CPP. Both of them are driven by AG\_MemAcc() and show how to spread the different memory spaces of the 8051 architecture.

## The Memory Interface

The AGDI interface uses the function AG\_MemAcc to access the target memory:

```
U32 _EXPO_ AG_MemAcc (U16 nCode, UC8 *pB, GADR *pA, UL32 nMany);
```

The parameter 'nCode' specifies the function sub-code:

**AG\_READ:** read memory

**AG\_WRITE:** write memory

**AG\_WROPC:** write code memory (used while loading a user program)

**AG\_RDOPC:** read code memory (used for disassembly)

Although both AG\_WROPC and AG\_RDOPC seem to duplicate AG\_WRITE and AG\_READ, they allow you to distinguish between code and non-code accesses. The sample code in AGDI.CPP contains both a code and data cache to provide fast access to memory when the target is not accessible. This is very important since for example the disassembly window makes many accesses to code memory especially if you hit the PageUp key.

The Parameter 'pB' is a pointer to buffer which is at least 'nMany' bytes in size. 'GADR' gives the memory address to access, as described under the section 'Address Representation'. Note that both members 'mSpace' and 'nLen' are not used. The 'nMany' parameter specifies the number of bytes to be transferred.

The AGDI memory interface uses one more function to speed up  $\mu$ Vision2 accesses to memory attributes:

```
U32 _EXPO_ AG_MemAtt (U16 nCode, UL32 nAttr, GADR *pA);
```

The story behind AG\_MemAtt is that  $\mu$ Vision2 requires fast access to memory attributes such as: location at address contains 'executable code' or has 'an enabled or disabled breakpoint' set on it and so on. This type of information is displayed in the gutter area (the gray colored section to the left) of all  $\mu$ Vision2's editor windows or the disassembly window and is heavily accessed. Fast access to the attributes is a very important issue here, otherwise it would take considerably time to redraw the content



of an editor window which in fact could cause the editor or disassembly window to become almost unusable.

The important sub-code is AG\_GETMEMATT, where  $\mu$ Vision2 requests the attribute segment. The sample code contains a code and data cache. It holds the memory values and the attributes of each memory location:

```
#define _MSGM (65536 + 4) // memory required for a single 64k segment
#define _ASGM _MSGM * 2 // attributes per memory location are 16 bits wide !

struct MMM {
    UC8 *mem; // Pointer to Memory Image
    U16 *atr; // Memory Attributes Segment
};
extern struct MMM mslots [3]; // [0]=data/idata, [1]=xdata/pdata, [2]=code
```

The SlotNo() function in AGDI.CPP maps addresses to one of the 3 cache slots.

Here are the possible attributes as defined in AGDI.H:

```
#define AG_ATR_EXEC 0x01 // location is 'executable'
#define AG_ATR_READ 0x02 // location is 'readable'
#define AG_ATR_WRITE 0x04 // location is 'writable'
#define AG_ATR_BREAK 0x08 // location has an 'Execution-Break' on it
#define AG_ATR_EXECD 0x10 // location has been 'Executed'
#define AG_ATR_WATCH 0x20 // Location has a Watch
#define AG_ATR_BPDIS 0x40 // 'disabled Exec-Break' Attribute
#define AG_ATR_PAP 0x80 // Location has a Perf.-Analyzer point
#define AG_ATR_WRBRK 0x100 // Loc has a write-access break
#define AG_ATR_RDBRK 0x200 // Loc has a read-access break
#define AG_ATR_EXTR 0x400 // 166/167: within range of an EXTR sequence
#define AG_ATR_JTAKEN 0x4000 // Jump at location was true executed
```

Note that not all of the given attributes need to be implemented. For a target driver, the attributes AG\_ATR\_EXEC, AG\_ATR\_BREAK and AG\_ATRBPDIS are important as this information is reflected in the editor views and the disassembly window.

---

**Note: Do not change the attribute definitions in AGDI.H in order to avoid collisions with the attribute values used in  $\mu$ Vision2 !**

---

The AG\_ATR\_EXEC attribute can be set for each location while an AG\_MemAcc with nCode AG\_WROPC is executed.

The AGDI.CPP source file contains the complete memory interface where all the read and write operations are spread into different functions such as ReadCode(), ReadXdata(), WriteData(), WriteSFR() and so on. The places where you should add your specific code are marked with '/\*---TODO:'.

The attribute AG\_ATR\_JTAKEN and AG\_ATR\_EXECD mark the current execution state of the instruction and is used to display the CODE COVERAGE information in the  $\mu$ Vision2 Debugger. The following colors are displayed depending on the bits:

AG\_ATR\_EXECD = 0, AG\_ATR\_JTAKEN = 0; color gray, instruction not executed at all  
AG\_ATR\_EXECD = 1, AG\_ATR\_JTAKEN = 0; color orange, conditional jmp to address never taken  
AG\_ATR\_EXECD = 0, AG\_ATR\_JTAKEN = 1; color cyan, conditional jmp to address always taken  
AG\_ATR\_EXECD = 1, AG\_ATR\_JTAKEN = 1; color green, instruction (cont. jmp) fully executed

---

**Note: Only conditional jump instructions set AG\_ATR\_EXECD and AG\_ATR\_JTAKEN individually. All other instructions set always both bits AG\_ATR\_EXECD and AG\_ATR\_JTAKEN.**

---

---

## Execute Program Code

uVision2 controls program execution with the AG\_GoStep function. AG\_GoStep gets the following nCode values:

```
nCode = AG_STOPRUN    // Stop Go/Step.
nCode = AG_NSTEP      // execute 'nSteps' instruction steps
nCode = AG_GOTILADR    // run til 'pA->Adr' or some Bp,
nCode = AG_GOFORBRK   // run forever or till some Bp reached.
```

AG\_GoStep runs in a separate thread and does not return until execution stops. Before AG\_GoStep is entered  $\mu$ Vision2 enables the Stop toolbar button. When AG\_GoStep returns the Stop button is disabled. When execution should be stopped (with the Stop toolbar button) AG\_GoStep is called again with nCode = **AG\_STOPRUN**. The sample implementation in AGDI.CPP sets up the address breakpoints before starting to execute the user program in the AG\_GOTILADR or AG\_GOFORBRK cases. After the execution stops, the breakpoints are removed.

You need to fill out the SetClrBp() function in AGDI.CPP to set and clear the address breakpoints. The details here depend on your target hardware. There are two more functions which you need to fill out, Step() and GoCmd(). Step() should perform a single instruction step. GoCmd() should start execution the user program until an address breakpoint is encountered or some stop criteria is detected.

When the Stop button on the toolbar is pressed, AG\_GoStep() is activated with the AG\_STOPRUN sub-code. At this point, you should stop execution of the user program.

## Breakpoints

When the user defines or modifies Breakpoints,  $\mu$ Vision2 calls the Target Driver function **AG\_BpInfo**:

```
U32 _EXPO_ AG_BpInfo (U16 nCode, void *vp);
```

This function does not set the breakpoint in the hardware, but allows the  $\mu$ Vision2 editor to query and change the current active breakpoints and to kill or disable all of them. The breakpoint information is stored using attribute segments, as described in the Memory Interface section. The AG\_BpInfo sub-codes are shown in the following list:

```
#define AG_BPQUERY      0x01    // Same as AG_BPEXQUERY
#define AG_BPTOGGLE     0x02    // Not used with target driver
#define AG_BPINSREM     0x03    // Not used with target driver
#define AG_BPACTIVATE   0x04    // Not used with target driver
#define AG_BPDISALL     0x05    // Notification: all address breaks need to be disabled
#define AG_BPKILLALL    0x06    // Notification: all address breaks need to be killed
#define AG_BPEXQUERY    0x07    // Get attributes: AG_ATR_BREAK, AG_ATR_BPDIS, AG_ATR_EXECD
#define AG_BPENABLE     0x08    // Notification: Enable Breakpoint at address
#define AG_BPDISABLE    0x09    // Notification: Disable Breakpoint at address
#define AG_BPKILL       0x0A    // Notification: Kill Breakpoint at address
#define AG_BPSET        0x0B    // Notification: Set Breakpoint at address
```

Take a look at the AG\_BpInfo() function in file AGDI.CPP for details.

Another function is the function **AG\_BreakFunc** which is very similar to AG\_BpInfo:

```
_EXPO_ AG_BP *AG_BreakFunc (U16 nCode, U16 n1, GADR *pA, AG_BP *pBp);
```

The differences are the sub-codes and in that it receives a pointer to a breakpoint definition, an AG\_BP structure. The function codes are:

- 0x01: a new breakpoint is linked
- 0x02: the breakpoint is unlinked, attribute should be updated
- 0x04: a breakpoint changes its enabled/disabled attribute

---

0x05: is breakpoint acceptable by your target hardware

The list of breakpoints is maintained by  $\mu$ Vision2. It should be noted that both functions AG\_BpInfo() and AG\_BreakFunc() should deal with the attributes only. The actual breakpoints should be setup in the target hardware just before the execution is started via AG\_GoStep AG\_GOTILADR or AG\_GOFORBRK.

## Extension Menus

By using AGDI, you can add your own menu entries to  $\mu$ Vision2's Peripherals menu. This is required if you want to have dialogs within the target driver.

You need to setup an array of 'DYMENU' structures. The AGDI.CPP file contains sample code for 2 menu entries, the first is the configuration dialog, the second one is for a modeless dialog:

```
static DYMENU Menu[] = {
{ 1, "Target Settings", ConfDisp, 0, 0, }, // modal dialog
{ 1, "Modeless Dialog", MdShow, 0, 0, &ModDlg }, // modeless dialog
{ -1, /* End of menu list */ },
};
```

Each menu entry consists of a delimiter code (1,2,-2,-1), the menu item title and the address of a function which shows or hides the dialog. For modeless dialogs, the menu entry has an additional descriptor called 'DIAD' structure which addresses a update and a kill dialog function.

The menu structure is registered in the AG\_Init() function, AG\_INITITEM with sub-code AG\_INITMENU. The sub-code AG\_INITEXTDLGUPD expects the address of the general dialog update function:

```
case AG_INITITEM: // init item
switch (nCode & 0x00FF) {
case AG_INITMENU: // init extension menu
*((DYMENU **) vp) = (DYMENU *) Menu;
break;

case AG_INITEXTDLGUPD: // init modeless extension dlg update function
*((UC8 **) vp) = (UC8 *) DlgUpdate;
break;
```

After a Single Step has been executed or a Go has been stopped,  $\mu$ Vision2 notifies all relevant windows and dialogs to update themselves to reflect the current values. In case of the target dialogs, it calls the DlgUpdate() function, as registered above.

Note that you also need a 'CloseAllDlg' function to close all currently open dialogs before the driver is shut down. The AGDI.CPP sample code contains all the required functions and data for the extensions.

When creating your own modeless dialogs, you should use the template code in TestDlg.cpp,h as a reference. This is important since these dialogs must use a different constructor and destructor code.

---

*Note: When creating your own dialog resources for a modeless dialog, make sure that the 'Visible' style is set, otherwise the dialog frame will always be invisible. Modeless dialogs are different in this aspect from modal dialogs.*

---

## Serial I/O

$\mu$ Vision2 supports two serial windows to simulate the serial input and output. The serial windows can be accessed by the target driver as well. If you want to display data in the serial window #1, the appropriate call would be:

```
char szSerTxt[] = "this should appear in serial window #1";
AG_Serial (AG_SERBOUT, 0, sizeof (szSerTxt) - 1, (void *) szSerTxt);
```

When the user presses a key in serial window #1 for example, then  $\mu$ Vision2 calls the AG\_Serial() function with nCode=AG\_SERXIN, nSerNo=0, nMany=1 and vp pointing to the character.

## AGDI Callbacks

AGDI can borrow some of the functionality from  $\mu$ Vision2. This can be done by using the 'pCbFunc' callback pointer, a function code and the appropriate parameters. Note that only the most important callback functions are described here. The additional ones can be found in AGDI.H near the end of the file.

### Execute a $\mu$ Vision2 debug command

Almost any command except 'Exit' can be executed.

Example: `pCbFunc (AG_CB_EXECCMD, "dir public");` // execute dir public symbols command

### Force $\mu$ Vision2 to update all Windows

Causes an update of all currently open debug windows and dialogs. The function returns when the updates are completed.

Example: `pCbFunc (AG_CB_FORCEUPDATE, NULL)`

### Change $\mu$ Vision2's Statusbar message string

Example: `pCbFunc (AG_CB_MSGSTRING, "Running...");` // up to 20 characters

## Disassemble Opcode, Assemble a single Instruction

AG\_CB\_DISASM can be used to disassemble the given opcodes or to assemble the given instruction.

Example for code disassembly:

```
DAAS    parms;

parms.Adr = (amCODE << 24) | 0x1000;    // disassemble address C:0x1000
parms.Opc[0] = 0x90;                    // MOV DPTR,
parms.Opc[1] = 0x12;
parms.Opc[2] = 0x34;                    //          #0x1234
parms.Opc[3] = 0;

pCbFunc (AG_CB_DISASM, (void *) &parms); // disassemble...

//--- on return:  parms.OpcLen := length of opcode in bytes
//---            parms.szB[]  := disassembled instruction in ascii
//--- Note:       parms.Adr    is used for ascii-address and relative branches only.
//---            parms.Result := always 0.
```

Example for inline Assemble:

```
DAAS    parms;
```

---

```
parms.Adr = (amCODE << 24) | 0x1000;    // assemble address C:0x1000
strcpy (parms.szB, "MOV DPTR,#0x1234"); // instruction to assemble

pCbFunc (AG_CB_INLASM, (void *) &parms); // assemble...

//--- on return:
//---  parms.Result      = 0 if successful, otherwise != 0
//---  parms.OpcLen := length of opcode in bytes
//---  parms.Opc[]  := 'parms.OpcLen' Opcode bytes
```