Micromega Corporation

Release Notes

uM-FPU64 IDE Release 411

Changes for IDE Release 411

uM-FPU64 IDE Release 411 adds several new features and fixes some known problems.

Firmware Upgrade

To use uM-FPU64 IDE r411 software, the uM-FPU64 chip must be running firmware release 411 or higher. Firmware files are supplied with the IDE and installed in the *Firmware* folder of the IDE installation directory. The firmware can be upgraded using the *Tools>Firmware Update…* menu item. Select the appropriate firmware file as follows:

28-pin chip:	*uMFPU64 64K28 Firmware V411.dat*
44-pin chip:	*uMFPU64 64K44 Firmware V411.dat*

Highlights

Parallax Propeller Support

New code generation and a new driver for the Parallax Propeller provide significantly enhanced performance for Propeller programs. The new capability for updating target files with linked code supports updating Propeller UTF-16 files (files that contain special characters).

Code Generation using Data Commands

New `WRITE_CMD`, `READ_CMD`, and `WRITE_DATA` commands have been added to the target description file. These commands allow the code generator to generate code using these new commands instead of the write byte commands. This creates more efficient code for targets such as the Parallax Propeller.

Updating Target Files with Linked Code

An automated method for updating target source files with code generated by the compiler has been added to the IDE using the ***Update Target File...*** button in the ***Output Window***. To use the automated update method, special comments are inserted into the target source file to define the begin and end points for code insertion. See additional information below.

Trace Messages for FPU Errors

If an error occurs on the FPU and the debug monitor is enabled, a trace message is now displayed, and Break occurs. See additional information below.

RAM Display window

The RAM Display window has been reimplemented with the following features:

- displays the memory allocation, including DMA area

- different allocations are colour coded
- changes to RAM since the last read are highlighted in red
- non-zero values are shown with a light yellow background
- format files can be loaded and saved
- descriptions can be interactively entered in the formatted display
- the format is saved on exit and restored when the IDE restarts
- values can be displayed in decimal, hex, binary and ASCII

Changes to IDE Interface

- new *RAM Display Window*
- fixed problem with spurious horizontal scroll bar in *Matrix Window*
- fixed Matrix Window Update button to update both register values and memory
- clicking in the upper left corner of the *Number Converter*, *Interactive Compiler*, *SERIN*, *SEROUT*, *Flash*, *RAM*, or *Matrix* windows will cause the main IDE window to be displayed immediately behind the current window.
- *Update Target File...* button added to *Output Window*
- fixed problem with displaying `{DEBUG ON}`, `{DEBUG OFF}` messages
- added trace messages for FPU `{ERROR:xxxx}` messages

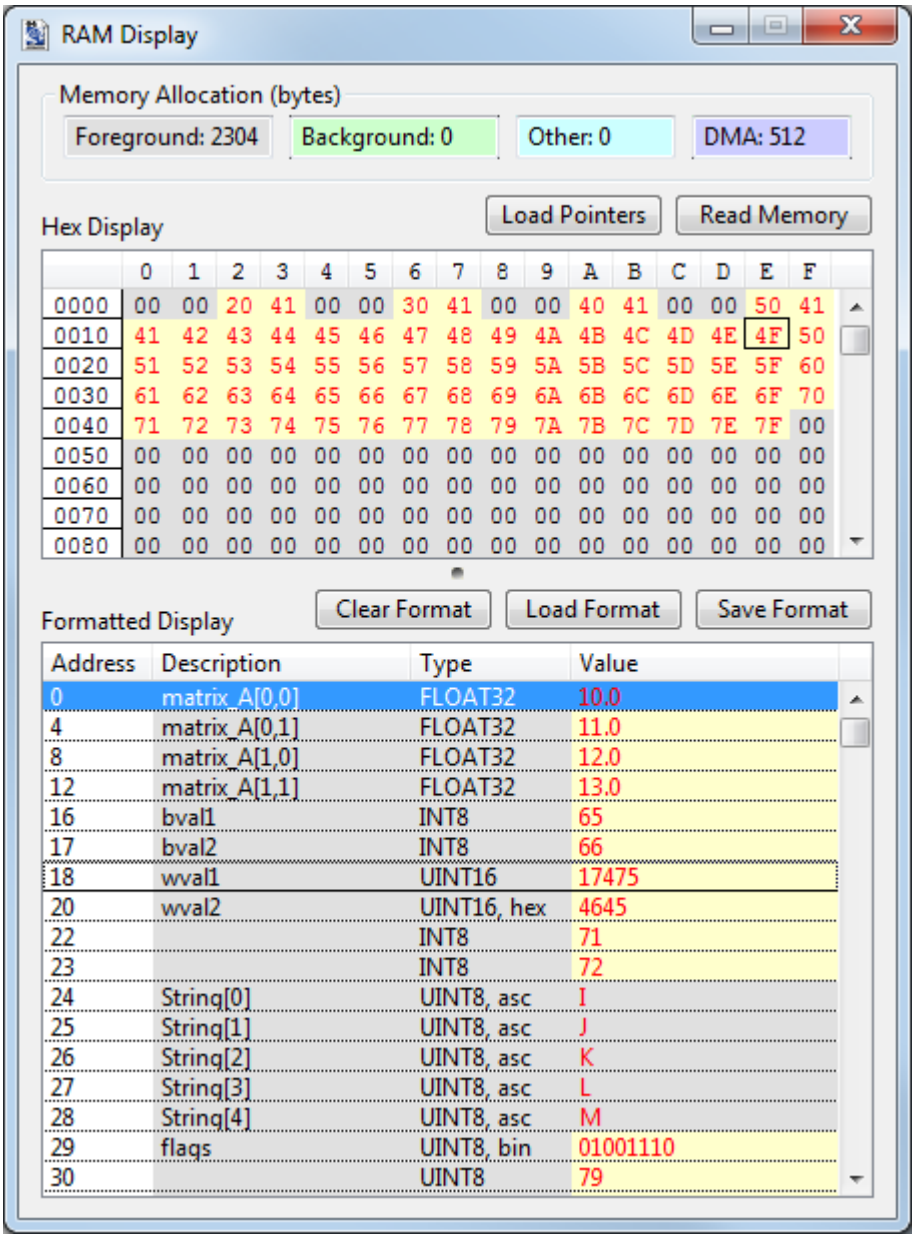
Changes to Compiler

- added definitions for `DEVIO,SDFAT`
- comments included on the FPU source line for register definitions, variable definitions, and function definitions are now included in the target code generated.
- added `devio(FIFOn, ALLOC_MEM, size)` function
- added support for `devio(FIFOn, ALLOC_MEMR, regSize)` function
- added `#DEVICE` *`device_file{,device_name}`* directive
- removed support for `devio(VDRIVE2,…)` functions
- added `#TARGET_OPTIONS, PROPELLER` directive
- added `#TARGET_CODE` *`link_ID`* directive
- fixed target code generation for assigning 64-bit float constants to a register

Changes to Target Description File

- added `TARGET_OPTIONS=<PROPELLER>`
- added `WRITE_CMD=<>` command
- added `READ_CMD=<>` command
- added `WRITE_DATA=<>` command

RAM Display Window



Memory Allocation shows the allocation of RAM to the various memory areas.

Foreground	Memory allocated to the foreground process.
Background	Memory allocated to the background process.
Other	Memory allocated to FIFO1, FIFO2, FIFO3, FIFO4, and any loadable devices.
DMA	DMA memory. Used by the ADC instructions. Can be accessed with indirect pointers.

The ***Load Pointers*** button set the description, type and value fields for any foreground pointer currently loaded in the Register display of the Debug window. If the pointer is an array pointer, each element of the array is added as a description.

The ***Read Memory*** button reads the current contents of RAM and updates the displays. If the memory allocation has changed, the formatted display is cleared, and the last format file used is reloaded. All RAM values that have changed since the last read are highlighted in red, and all non-zero values are shown with a light yellow background.

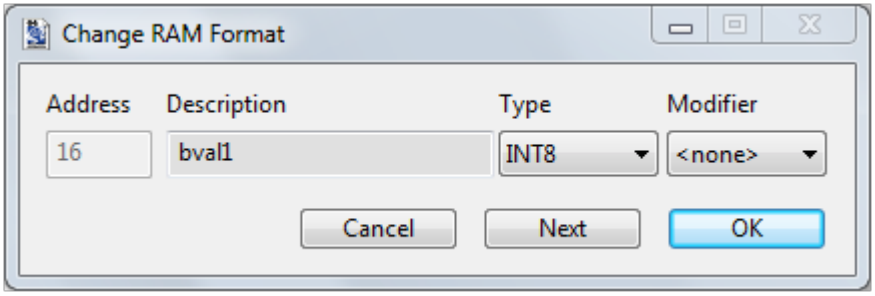
The ***Clear Format*** button clears the formatted display. If the RAM format file *default.txt* exists in the *~/My Documents/Micromega/RAM Files* folder it will be loaded and the formatted display is updated.

The ***Load Format*** button loads a RAM format file and updates the formatted display.

The ***Save Format*** button saves a RAM format file.

The ***Hex Display*** shows the value of each byte in RAM as a hexadecimal value. The current selection in the formatted display outlined with a box in the hex display. Clicking in the hex display will select the corresponding item in the formatted display. Values that have changed since the last time RAM was read are highlighted in red, and non-zero value are shown with a light yellow background.

The ***Formatted Display*** shows the RAM contents formatted according to the type specified. Each row in the formatted display can have a separate description, type, and modifier. The description, type and modifer can be entered using a RAM format file, or entered interactively using the *Change RAM Format* dialog that is displayed by right-clicking on a row in the formatted display. Multiple rows can be changed by first selecting the multiple rows, then right-clicking within the selection.



The Description field can be used to enter any text string that doesn't include a double quote (") character. There are some special cases:

n	The type and modifier will be repeated *n* times specified (where *n* is a decimal number).
*	The type and modifier will be repeated until the end of the memory area.
name[i] *name[i, j]* *name[i, j, k]*	Specifies an array *name*, with the dimensions of the array given by *i*, *j* and *k*. For each element of the array, the description will

be set the the name of the element and the type and modifier will be repeated.

If you wish to use one of the special cases as a description, without it being handled as a special case, then the *description* should be enclosed in double quotes (“). (e.g. `"name[2,2,2]"` will not be expanded into multiple array elements).

RAM Format Files are text files containing a description of the format to use in the formatted display. They are stored in the *~/My Documents/Micromega/RAM Files* folder. The *autosave.txt* file is saved automatically to the *~/My Documents/Micromega/RAM Files* folder when the RAM Display window is closed, and loaded when the RAM Display window is first opened. The RAM format file *default.txt* is used to specify the default format for the formatted display. If *default.txt* exists in the *~/My Documents/Micromega/RAM Files* folder it will be loaded when the *Clear Format* button is pressed. Other files can be written and edited by the user. The RAM format files can contain the following lines:

Header

`<RAM FORMAT>` or `<RAM FORMAT OVERLAY>`

This must be the first line of the file. The `<RAM FORMAT>` line indicates that the file contains a full format description. The formatted display is cleared before loading the format file. The `<RAM FORMAT OVERLAY>` line indicates that the file is an overlay. The descriptions and types defined in the file will be added the existing formatted display.

Comment

`; comment`

Any line that begins with a semi-colon (;) is a comment line. The *autosave.txt* file adds comments showing the date and time and the memory allocation in effect when the file was saved.

Memory Area

`<FOREGROUND>`
`<BACKGROUND>`
`<DMA>`
`<FIFO1>` to `<FIFO4>`
`<DEVICE1>` to `<DEVICE6>`

Specifies the memory area for the description lines that follow. An optional offset can be added as a second argument (e.g. `<FOREGROUND, 100>`). This specifies a decimal offset into the memory area for the next description line. The offset can also have multiple decimal values that are added together (e.g. `<FOREGROUND, 100+10>`).

Description

`description, type, modifier`

The *description* can be any text string that doesn't include a double quote (") character. There are some special cases:

n	The *type* and *modifier* will be repeated *n* times specified (where *n* is a decimal number).
*	The *type* and *modifier* will be repeated until the end of the memory area.
name[i] *name[i, j]* *name[i, j, k]*	Specifies an array *name*, with the dimensions of the array given by *i*, *j* and *k*. For each element of the array, the description will be set the the name of the element and the type and modifier will be repeated.

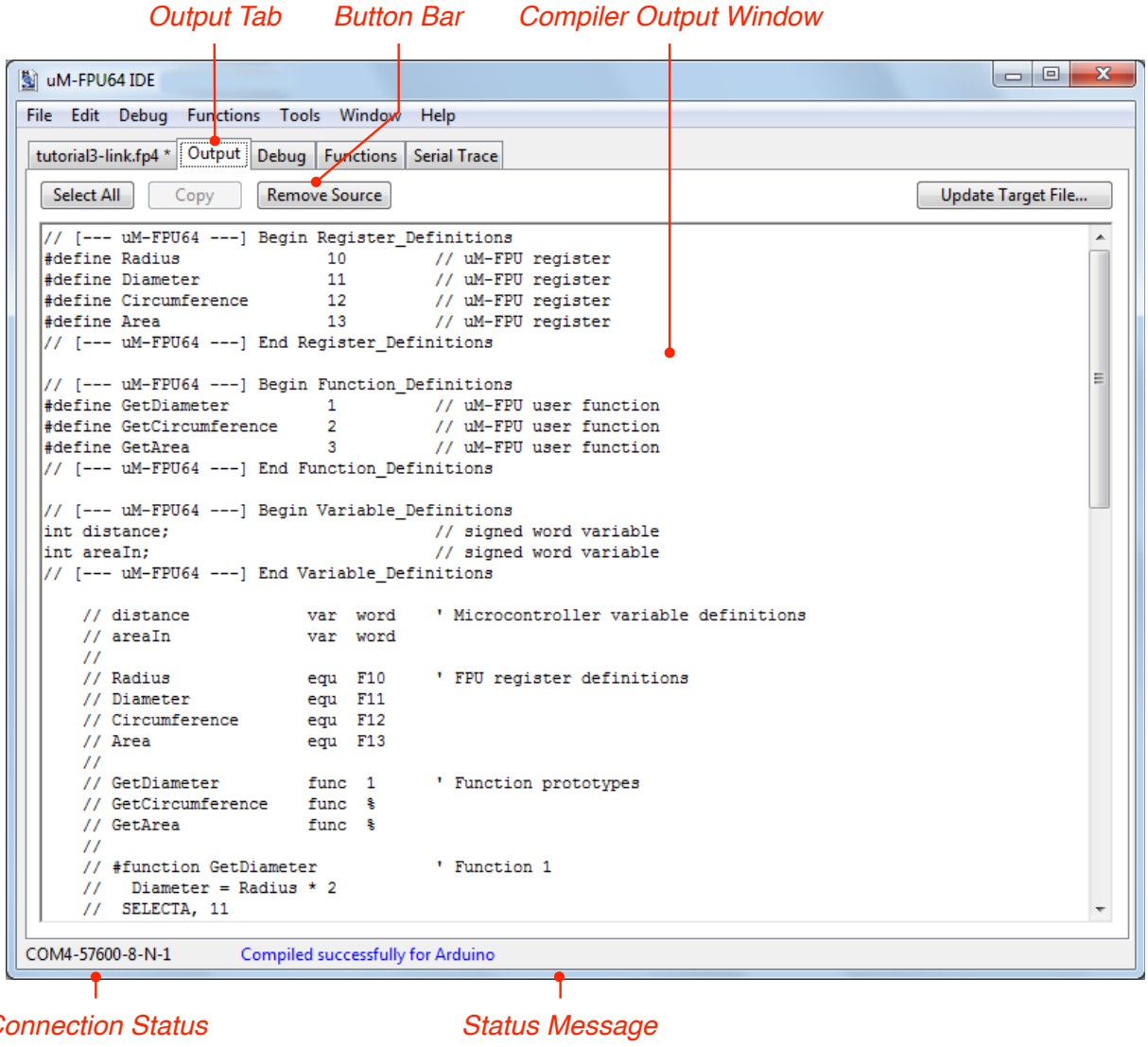
If you wish to use one of the special cases as a description, without it being handled as a special case, then the *description* should be enclosed in double quotes ("). (e.g. `"name[2,2,2]"` will not be expanded into multiple array elements).

If the *description* string contains a comma, or you wish to use one of the special cases without it being handled as a special case, then the *description* must be enclosed in double quotes ("). (e.g. `"name[2,2,2]"` will not be expanded into multiple array elements).

The *type* can be one of the following: `INT8`, `UINT8`, `INT16`, `UINT16`, `LONG32`, `ULONG32`, `FLOAT32`, `LONG64`, `FLOAT64`.

The *modifier* is optional, and if not specified no modifier is used. The modifier can be one of the following: `HEX`, `BIN`, `ASC`. The `BIN` modifier only displays the lower 16 bits if the *type* is greater than 16 bits. The `ASC` modifier displays the ASCII value of the lower 8 bits.

Updating Target Files with Linked Code



Target code generated by the compiler can by manually copied to target source files using copy-and-paste. An automated update method is available using the ***Update Target File...*** button in the ***Output Window***. To use the automated update method, special comments are inserted into the target source file to define the begin and end points for code insertion. These special comments, or links, are generated by the compiler. Links are automatically generated for register definitions, function definitions, and variable definitions. An example of a register definition link is shown below:

```
// [--- uM-FPU64 ---] Begin Register_Definitions
#define Radius                10          // uM-FPU register
#define Diameter              11          // uM-FPU register
#define Circumference         12          // uM-FPU register
#define Area                  13          // uM-FPU register
// [--- uM-FPU64 ---] End Register_Definitions
```

Other user-defined links are generated by using the ***#target_code link_id*** directive in the FPU file. For example, using the following directive in the FPU file:

```
#target_code calculations
```

Will generate the following code in the ***Output Window***.

```
// [--- uM-FPU64 ---] Begin calculations
  //   Radius = distance / 1000  ' Calculations
  Fpu.write(SELECTA, Radius, LOADWORD);
  Fpu.writeWord(distance);
  Fpu.write(FSET0, LOADWORD);
  Fpu.writeWord(1000);
  Fpu.write(FDIV0);
// [--- uM-FPU64 ---] End calculations
```

To initially insert links into the target source file, copy-and-paste the links from the ***Output Window*** to the target source file.

When the ***Update Target File...*** button is pressed, a dialog is displayed so the user can select a target file. Any link in the target file with a matching link in the ***Output Window*** will be updated with the code from the ***Output Window***. A timestamp comment is added to the start of the linked code stored in the target file. Linked code is inserted into the target file using the indentation of the begin link in the target file. This allows the inserted code to be properly aligned with other target code.

Trace Messages for FPU Errors

If an error occurs on the FPU and the debug monitor is enabled, a trace message is now displayed, and Break occurs. The error messages are as follows:

FPU Error: Address error

An address error occurred inside an XOP instruction. The likely cause is an invalid parameter being specified in an XOP instruction.

FPU Error: Buffer overflow

The 256 byte FPU instruction buffer has been exceeded. This can be avoided by waiting for a ready status at least every 256 bytes, if more than 256 byes are sent to the FPU between read operations. If debug trace is enabled, instructions take longer to execute, particularly if the serial buffer fills, which can sometimes lead to an FPU buffer overflow that doesn't occur an normal execution speed.

FPU Error: Call level exceeded

The 16 levels of call nesting available on the uM-FPU64 has been exceeded.

FPU Error: Device not loaded

A `DEVIO,`*`device`*`,LOAD_DEVICE,…` instruction failed because the loadable device was not programmed into Flash memory.

FPU Error: Function not defined

A user function has been called that has is not currently stored in FPU Flash memory.

FPU Error: Incomplete Instruction

An instruction that requires multiple bytes has not received the required number of bytes within the timeout period of one second. This is generally caused by a programming error in the target code.

FPU Error: Invalid parenthesis

There are 8 levels of parentheses available using the `LEFT` and `RIGHT` instructions. Either too many `LEFT` instructions have been sent, or there is a mismatch with the number of `LEFT` and `RIGHT` instructions.

FPU Error: Memory Allocation failed

A memory allocation failed because the number of bytes requested were not available in the dynamic allocation area.

FPU Error: XOP not defined

An extended opcode (`XOP`) was called that is not currently stored in FPU Flash memory.

Compiler Changes

#DEVICE

Defines a loadable device. The device code is loaded from the specified device library file.

Syntax

#XOP device_file{:device_name}

Name	Type	Description
device_file	*string*	Specifies the name of a Device Library File.
device_name	*string*	Specifies the name of the loadable device. If *device_name* is not specified, then *device_name* is the same as the *device_file* name.

The device code is loaded from the specified device library file. A `#DEVICE` directive and a call to `devio(device,LOAD_DEVICE,device_name)` must be included in the FPU source file before a loadable device can be used.

Examples

```
#DEVICE sdfat        ; loads the SD FAT16/FAT32 device
```

#TARGET_CODE

Specify target code link.

Syntax

#TARGET_CODE *link_ID*

Name	Type	Description
link_ID	*string*	Specifies a unique link ID.

This directive instructs the compiler to generate a target code link. Target code links allow target files to be automatically updated when the *Update Target File…* button is pressed in the *Output Window*. A begin link is output at the start of linked code and an end link is output at the end of the linked code. The user can define as many links as needed.

Example

This source code:

```
#TARGET_CODE Section1
 F1 = F2 + 10
```

Generates the following output for the Arduino target:

```
// [--- uM-FPU64 ---] Begin Section1
  // F1 = F2 + 10
  Fpu.write(SELECTA, 1, FSET, 2, FADDI, 10);
  //
// [--- uM-FPU64 ---] End Section1
```

If this linked code is copied to the target file, output from future compiles can be automatically copied to the target file using the *Update Target File…* button in the *Output Window*.

Target Description File Changes

READ_CMD — Define format for Read Command Instructions

`READ_CMD=<string>`
Default: empty string
Parameters: {name},{n},{data}
Example: `READ_CMD=<{name} := FPU.ReadCmd{n}({data}>`

Description: This command defines the format of the read command instructions. Parameter {name} is replaced with the target variable name. Parameter {n} is replaced with the appropriate read command as follows:

```
ReadCmd, ReadCmdByte, ReadCmdWord, ReadCmdLong, ReadCmd2Long,
ReadCmdStr, ReadCmdByte3.
```

Parameter {data} is replaced with the appropriate data items for the read command.

WRITE_CMD — Define format for Write Command Instructions

`WRITE_CMD=<string>`
Default: empty string
Parameters: {n},{data}
Example: `WRITE_CMD=<FPU.WriteCmd{n}({data}>`

Description: This command defines the format of the write command instructions. Parameter {n} is replaced with the appropriate write command as follows:

```
WriteCmd, WriteCmdByte, WriteCmdByte2, WriteCmdByte3,
WriteCmdByte4, WriteCmdByteWord, WriteCmdByte2Word,
WriteCmdByte2Word, WriteCmdByteLong, WriteCmdWord,
WriteCmdLong, WriteCmdStr, WriteCmdByteStr, WriteCmdByte2Str.
```

Parameter {data} is replaced with the appropriate data items for the write command.

WRITE_DATA — Define format for Write Data Instructions

`WRITE_DATA=<string>`
Default: empty string
Parameters: {n},{data}
Example: `WRITE_DATA=<FPU.WriteData{n}({data}>`

Description: This command defines the format of the write data instructions. Parameter {n} is replaced with the appropriate write data suffix depending on the number of data items.

```
WriteData1, WriteData2, WriteData3, …
```

Parameter {data} is replaced with the appropriate data items for the write data instructions. The write data instructions have a datatype as the first argument, which specifies the data types of the remaining arguments. This is used by targets that pass all arguments as 32-bit values (e.g. Parallax Propeller). The code generator create the datatype value as a series of is a 2-bit values:

00 8-bit data
01 16-bit data

10 32-bit data
11 8-bit opcode

The datatype bits are specified from left to right, and the value is right justified.