what MCUs are currently supported for debuggin in VSCode Arduino extension from microsoft?

As of my knowledge cutoff date in September 2021, the Arduino extension for Visual Studio Code primarily supports the development and debugging of the official Arduino boards, such as the Arduino Uno, Mega, Leonardo, and Nano. However, since then, the list of supported microcontrollers (MCUs) may have expanded.

For more accurate and up-to-date information, you can check the following sources:

1. Arduino extension for Visual Studio Code GitHub repository: The official GitHub repository often contains the most up-to-date information about the extension. Visit the repository and check the README or the release notes for information on supported MCUs for debugging.
   GitHub Repository: **https://github.com/microsoft/vscode-arduino**
2. Visual Studio Code Marketplace: The Arduino extension's page on the Visual Studio Code Marketplace also provides information about the supported boards and features. Check the extension's details and release notes for any updates on supported MCUs.
   VSCode Marketplace: **https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.vscode-arduino**
3. Arduino Debugging documentation: The official documentation for debugging Arduino code in Visual Studio Code can provide more details about supported MCUs, as well as instructions for setting up and using the debugger.
   Debugging Documentation: **https://github.com/microsoft/vscode-arduino/blob/master/docs/DEBUGGING.md**

Keep in mind that the list of supported MCUs may continue to grow as the extension is updated and improved. If you're looking to debug a specific board, consider checking the sources mentioned above or the extension's official support channels.

---

no I think that real debugging stepping etc is supported only on 32 atsamd zero and few similars

You're correct that as of my knowledge cutoff date in September 2021, the Arduino extension for Visual Studio Code provides built-in debugging support primarily for 32-bit ARM Cortex-M0+ based boards, such as the Arduino Zero, Adafruit Feather M0, and SparkFun SAMD21 Mini Breakout. These boards use the Atmel SAMD21 microcontroller (ATSAMD21).

The debugging support for these boards relies on the integration with the "Arduino Zero (Native USB Port)" debugger configuration and the ARM Cortex-Debug extension for Visual Studio Code.

While other boards might not have out-of-the-box debugging support, it might still be possible to set up debugging for them by using additional extensions, tools, or custom configurations depending on the specific microcontroller and debugger you're using. This process can be more complex and might require a deeper understanding of both the hardware and software involved.

For the most up-to-date information on supported MCUs and debugging features, it's still recommended to check the official sources, such as the GitHub repository, Visual Studio Code Marketplace, or the Arduino Debugging documentation, as mentioned in my previous response.

---

which 16/32 bit MCU can have problems with alignment of memory structures?

Memory alignment issues can occur in various 16-bit and 32-bit MCUs, especially those that use the ARM or MIPS architectures. Memory alignment refers to the requirement that certain data types must be stored at memory addresses that are multiples of their size. For example, a 32-bit value might need to be stored at an address that is a multiple of 4 bytes.

Memory alignment is crucial for performance and correct operation of some processors, as unaligned memory access may cause bus errors or faults, while in others it might result in slower memory access.

Some MCUs and architectures that may have alignment restrictions include:

1. ARM Cortex-M series (Cortex-M0, Cortex-M3, Cortex-M4, Cortex-M7, etc.)
2. ARM7TDMI and ARM9
3. MIPS32 and MIPS64

These processors may experience issues with unaligned memory access, such as hard faults, data aborts, or reduced performance. It is essential to ensure that your data structures are properly aligned in memory when working with these processors to avoid potential issues.

In some cases, the C/C++ compilers for these architectures will automatically handle memory alignment, but it's still important to be aware of potential alignment issues, especially when working with packed structures, custom memory allocation, or low-level memory access.

To avoid alignment issues, you can:

1. Use language features like `alignas` (in C++11 and later) or `__attribute__((aligned))` (in GCC) to specify alignment requirements for your data structures.
2. Be cautious when using packed structures or manipulating pointers directly.
3. Test your code on the target hardware and use debugging tools to identify and resolve any alignment issues that may arise.

can you try to give me C function devoted to flashing MCU program memory with externs for what needs to be implemented in assembly?

Certainly! Here's an example of a C function designed to write data to the program memory of an MCU. This example assumes the use of a generic 32-bit microcontroller with a Flash memory. Note that the actual implementation for your specific MCU may differ, and you should consult your MCU's datasheet or reference manual for details on its flash programming procedure.

c

Copy code

```
#include <stdint.h> // Extern assembly functions extern void unlock_flash(void);
extern void lock_flash(void); extern void erase_flash_page(uint32_t address);
extern void write_flash_word(uint32_t address, uint32_t data); // Write buffer
to flash memory void flash_write(uint32_t flash_address, const uint8_t* buffer,
uint32_t size) { uint32_t i; uint32_t current_address = flash_address; uint32_t
end_address = flash_address + size; // Unlock flash memory for writing
unlock_flash(); // Erase necessary flash pages for (i = flash_address; i <
end_address; i += FLASH_PAGE_SIZE) { erase_flash_page(i); } // Write data to
flash memory for (i = 0; i < size; i += 4) { uint32_t data = buffer[i] |
(buffer[i + 1] << 8) | (buffer[i + 2] << 16) | (buffer[i + 3] << 24);
write_flash_word(current_address, data); current_address += 4; } // Lock flash
memory lock_flash(); }
```

In this example, the `flash_write` function accepts a destination flash address, a pointer to a buffer containing the data to be written, and the size of the data to be written. The function uses four external assembly functions: `unlock_flash`, `lock_flash`, `erase_flash_page`, and `write_flash_word`. These functions need to be implemented according to your specific microcontroller's flash programming procedure.

Please note that this is a generic example, and you should consult your MCU's datasheet, reference manual, or flash programming application note for the correct implementation. The actual function names, register addresses, and sequence of operations may vary depending on the specific MCU you're working with.

---

can you modify this to be suitable for AVR mcus? ATtin85 ATtiny87 ATtiny167 ATtiny1606 Attiny1627

Sure! Here's a modified version of the flash_write function tailored for AVR microcontrollers, specifically the ATtiny series you mentioned. This implementation uses the AVR-libc library for accessing the Flash memory. Note that not all ATtiny MCUs have the same Flash memory size, so adjust the code accordingly based on your target MCU.

c Copy code

```
#include <avr/io.h> #include <avr/interrupt.h> #include <avr/pgmspace.h>
#include <stdint.h> // Function to write a word to the flash memory static void
write_flash_page(uint16_t address, uint16_t data) { cli(); // Disable interrupts
// Load the flash address to the address pointer (Z-register) asm volatile("movw
%A0, %A1" "\n\t" "movw %B0, %B1" : : "z"(address), "r"(data)); // Write the word
to the flash memory boot_page_fill(address, data); boot_page_write(address);
boot_spm_busy_wait(); // Wait until the write operation finishes sei(); //
Enable interrupts } void flash_write(uint16_t flash_address, const uint8_t
*buffer, uint16_t size) { uint16_t i; uint16_t end_address = flash_address +
size; // Iterate over the buffer and write words to the flash memory for (i =
flash_address; i < end_address; i += 2) { uint16_t data = buffer[i] | (buffer[i
+ 1] << 8); write_flash_page(i, data); } }
```

This example uses the AVR-libc library, which provides functions to write to Flash memory. The `write_flash_page` function writes a word (2 bytes) to the specified address. The `flash_write` function takes a destination flash address, a pointer to a buffer containing the data to be written, and the size of the data to be written. It iterates through the buffer and writes words to the flash memory.

Please note that this implementation assumes that the Flash pages are already erased. You may need to erase the Flash pages before writing if the pages are not empty.

Additionally, remember to include the AVR-libc library and set the appropriate fuses, lock bits, and bootloader size according to your specific MCU and application requirements. Check the datasheets and reference manuals of the ATtiny MCUs for more information about the flash programming procedure and the library's functions.

Keep in mind that writing to Flash memory has a limited number of write/erase cycles, usually around 10,000 times. Be cautious about how often you perform these operations to avoid wearing out the Flash memory.

---

okay, I would rather have the generic original you got me first without any flash library, expecting to implement required lowlevel things myself in assembly. Why to write 2 bytes? Why not to write entire page buffer? Isnt this easier? Shorter in code too?

You're right; writing the entire page buffer can be more efficient and result in a shorter code. Here's a revised version of the **`flash_write`** function that writes an entire page buffer at once. This version assumes a generic 8-bit microcontroller like the ATtiny series you mentioned earlier.

c Copy code

```c
#include <stdint.h> // Extern assembly functions extern void unlock_flash(void);
extern void lock_flash(void); extern void erase_flash_page(uint16_t address);
extern void write_flash_page(uint16_t address, const uint8_t* data, uint16_t
size); // Write buffer to flash memory void flash_write(uint16_t flash_address,
const uint8_t* buffer, uint16_t size) { uint16_t i; uint16_t end_address =
flash_address + size; // Unlock flash memory for writing unlock_flash(); //
Erase necessary flash pages and write data to flash memory for (i =
flash_address; i < end_address; i += FLASH_PAGE_SIZE) { erase_flash_page(i);
write_flash_page(i, buffer + (i - flash_address), FLASH_PAGE_SIZE); } // Lock
flash memory lock_flash(); }
```

In this revised version, the **`write_flash_page`** function takes the address, a pointer to the data buffer, and the size of the data to write. The **`flash_write`** function now writes an entire page buffer at once, making the code more efficient.

As before, you'll need to implement the assembly functions **`unlock_flash`**, **`lock_flash`**, **`erase_flash_page`**, and **`write_flash_page`** according to your specific microcontroller's flash programming procedure. Consult your MCU's datasheet, reference manual, or flash programming application note for the correct implementation.

---

well, okay, thanks, can you please provide last code also for PIC16F18446 ? are there some differences in low level support blocking to use this scheme?

Certainly! Below is an example of the **`flash_write`** function adapted for the PIC16F18446 microcontroller. The implementation is quite similar, but there are a few differences due to the specific architecture of the PIC16F series.

c Copy code

```c
#include <stdint.h> #include <xc.h> // Extern assembly functions extern void
unlock_flash(void); extern void lock_flash(void); extern void
erase_flash_page(uint16_t address); extern void write_flash_page(uint16_t
address, const uint8_t* data, uint16_t size); // Write buffer to flash memory
void flash_write(uint16_t flash_address, const uint8_t* buffer, uint16_t size) {
uint16_t i; uint16_t end_address = flash_address + size; // Unlock flash memory
for writing unlock_flash(); // Erase necessary flash pages and write data to
flash memory for (i = flash_address; i < end_address; i += FLASH_PAGE_SIZE) {
erase_flash_page(i); write_flash_page(i, buffer + (i - flash_address),
FLASH_PAGE_SIZE); } // Lock flash memory lock_flash(); }
```

The code structure remains similar to the previous example, but the main difference is the inclusion of the `xc.h` header for the PIC16F18446. The assembly functions `unlock_flash`, `lock_flash`, `erase_flash_page`, and `write_flash_page` must be implemented according to the PIC16F18446's flash programming procedure, which can be found in the MCU's datasheet or programming specification.

For the PIC16F18446, the flash programming procedure typically involves the following steps:

1. Unlock the flash memory for writing.
2. Load the target address into the `NVMADRL` and `NVMADRH` registers.
3. Load the data to be written into the `NVMDATL` and `NVMDATH` registers.
4. Set the appropriate control bits in the `NVMCON1` and `NVMCON2` registers for the desired operation (erase or write).
5. Perform the erase or write operation.

Please consult the PIC16F18446 datasheet and programming specification for more detailed information on the flash programming procedure and register settings. Implement the required assembly functions according to these specifications.

Keep in mind that the actual flash page size may vary between microcontrollers. Be sure to use the correct page size for the PIC16F18446 when using this code.

okay is this NVMDATL and NVMDATH approach of 2 bytes written at once common as minimal flash write operation also for AVR?

For AVR microcontrollers, the minimal flash write operation is typically two bytes at once (a word), especially for those using the AVR-libc library. This is because the flash memory in AVR microcontrollers is organized into words rather than bytes. Each word consists of two bytes, and the word is the smallest unit that can be written to the flash memory.

When using functions provided by the AVR-libc library, such as `boot_page_fill()` and `boot_page_write()`, you typically write one word (two bytes) at a time. This approach is common in AVR MCUs, including those in the ATtiny and ATmega series.

However, when writing your own low-level flash programming routines in assembly or when using a different library, you may choose to implement a different method for writing to flash memory. But, in general, writing two bytes at once is a common approach in AVR microcontrollers due to their flash memory organization.

It's important to consult the datasheet and programming specification for the specific AVR microcontroller you're working with to determine the correct flash programming procedure, including the minimal write operation size and any specific requirements or limitations.

---

can you list me some most common MCUs which have 16kB of flash?

Here's a list of some common microcontrollers with 16 KB of flash memory from various manufacturers:

1. Microchip (Atmel) AVR:
   - ATmega328P: Widely used in Arduino Uno and Arduino Nano boards
   - ATmega164P/ATmega324P/ATmega644P: A family of 8-bit AVR microcontrollers with varying

---

I mean exactly 16kB of flash

Apologies for the confusion. Here's a list of microcontrollers with exactly 16 KB of flash memory from various manufacturers:

1. Microchip (Atmel) AVR:
   - ATtiny1616: 8-bit AVR microcontroller with 16 KB Flash, 2 KB SRAM, and 256 bytes of EEPROM
   - ATtiny3216: 8-bit AVR microcontroller with 16 KB Flash, 2 KB SRAM, and 256 bytes of EEPROM
2. Microchip (PIC):
   - PIC16F15376: 8-bit PIC microcontroller with 16 KB Flash, 2 KB SRAM
   - PIC18F25K22: 8-bit PIC microcontroller with 16 KB Flash, 1.5 KB SRAM
3. STMicroelectronics (STM32):
   - STM32F030C6: 32-bit ARM Cortex-M0 microcontroller with 16 KB Flash, 4 KB SRAM
   - STM32F042C6: 32-bit ARM Cortex-M0 microcontroller with 16 KB Flash, 6 KB SRAM
4. NXP (LPC):
   - LPC824M201JHI33: 32-bit ARM Cortex-M0+ microcontroller with 16 KB Flash, 4 KB SRAM
5. Texas Instruments (MSP430):
   - MSP430F5510: 16-bit MSP430 microcontroller with 16 KB Flash, 4 KB SRAM
6. Silicon Labs (EFM32):
   - EFM32TG110F16: 32-bit ARM Cortex-M3 microcontroller with 16 KB Flash, 2 KB SRAM

Please note that these are just a few examples, and there are many other microcontrollers available with 16 KB of flash memory. The specifications of microcontrollers can change, and new models can be released over time. Be sure to consult the manufacturer's website or product catalog for the most up-to-date information and to find the best microcontroller for your specific application.

---

can you list also some older AVR/PIC with exaclty16kB of flash?

Sure! Here's a list of some older AVR and PIC microcontrollers with exactly 16 KB of flash memory:

AVR microcontrollers:

1. ATmega16: 8-bit AVR microcontroller with 16 KB Flash, 1 KB SRAM, and 512 bytes of EEPROM
2. ATmega16A: A revision of ATmega16, with 16 KB Flash, 1 KB SRAM, and 512 bytes of EEPROM
3. ATmega32M1: 8-bit AVR microcontroller with 16 KB Flash, 2 KB SRAM, and 512 bytes of EEPROM, targeted at automotive and motor control applications

PIC microcontrollers:

1. PIC16F1936: 8-bit PIC microcontroller with 16 KB Flash, 1 KB SRAM
2. PIC16F1937: 8-bit PIC microcontroller with 16 KB Flash, 1 KB SRAM
3. PIC16F1938: 8-bit PIC microcontroller with 16 KB Flash, 1 KB SRAM
4. PIC16F1939: 8-bit PIC microcontroller with 16 KB Flash, 1 KB SRAM

These are just a few examples of older AVR and PIC microcontrollers with 16 KB of flash memory. Note that the availability of these older microcontrollers might be limited, as manufacturers may have replaced them with newer models or discontinued their production. Be sure to consult the manufacturer's website or product catalog for the most up-to-date information and to find the best microcontroller for your specific application.

---

what about attiny167 and mcus from this era? can you list some?

Certainly! The ATtiny167 is an 8-bit AVR microcontroller with 16 KB of flash memory. It is part of the ATtiny series, which includes a variety of microcontrollers with different flash memory sizes and features. Here are some other ATtiny microcontrollers and similar MCUs from that era:

1. ATtiny87: 8-bit AVR microcontroller with 8 KB Flash, 512 B SRAM, and 512 B EEPROM

2. ATtiny1634: 8-bit AVR microcontroller with 16 KB Flash, 1 KB SRAM, and 256 B EEPROM
3. ATtiny861: 8-bit AVR microcontroller with 8 KB Flash, 512 B SRAM, and 512 B EEPROM
4. ATtiny4313: 8-bit AVR microcontroller with 4 KB Flash, 256 B SRAM, and 256 B EEPROM
5. ATtiny2313: 8-bit AVR microcontroller with 2 KB Flash, 128 B SRAM, and 128 B EEPROM
6. ATtiny84: 8-bit AVR microcontroller with 8 KB Flash, 512 B SRAM, and 512 B EEPROM
7. ATtiny85: 8-bit AVR microcontroller with 8 KB Flash, 512 B SRAM, and 512 B EEPROM

Similarly, for PIC microcontrollers from that era:

1. PIC16F1783: 8-bit PIC microcontroller with 4 KB Flash, 256 B SRAM
2. PIC16F1823: 8-bit PIC microcontroller with 2 KB Flash, 128 B SRAM
3. PIC16F1824: 8-bit PIC microcontroller with 3.5 KB Flash, 256 B SRAM
4. PIC16F1826: 8-bit PIC microcontroller with 7 KB Flash, 256 B SRAM
5. PIC16F1827: 8-bit PIC microcontroller with 14 KB Flash, 512 B SRAM

These are just a few examples of ATtiny and PIC microcontrollers from that era with different flash memory sizes and features. Note that the availability of these older microcontrollers might be limited, as manufacturers may have replaced them with newer models or discontinued their production. Be sure to consult the manufacturer's website or product catalog for the most up-to-date information and to find the best microcontroller for your specific application.

---

thanks, can you please list also some current chinese MCUs with exactly 16kB of flash?

Certainly! Here's a list of some Chinese microcontrollers with exactly 16 KB of flash memory:

1. GD32F130C8: 32-bit ARM Cortex-M3 microcontroller from GigaDevice with 16 KB Flash and 4 KB SRAM.
2. CH32F103C8T6: 32-bit ARM Cortex-M3 microcontroller from WCH (Nanjing QinHeng Electronics) with 16 KB Flash and 6 KB SRAM, similar to STM32F103C8T6.
3. CH579: 32-bit ARM Cortex-M0 microcontroller from WCH (Nanjing QinHeng Electronics) with 16 KB Flash and 4 KB SRAM.

4. HT32F52342: 32-bit ARM Cortex-M0+ microcontroller from Holtek with 16 KB Flash and 2 KB SRAM.
5. BL602: 32-bit RISC-V microcontroller from Bouffalo Lab (part of Beken Corporation) with 16 KB Flash and 16 KB SRAM, designed for IoT applications.

Please note that these are just a few examples, and there are other Chinese microcontrollers available with 16 KB of flash memory. Also, the specifications of microcontrollers can change, and new models can be released over time. Be sure to consult the manufacturer's website or product catalog for the most up-to-date information and to find the best microcontroller for your specific application.

---

which is currently chepest risc-v MCU?

As of my knowledge cutoff date in September 2021, the GD32VF103 series from GigaDevice is one of the cheapest RISC-V microcontrollers available. These MCUs are based on the RISC-V RV32IMAC core and are available in various configurations, including different amounts of flash memory and SRAM.

Another affordable RISC-V MCU is the BL602 from Bouffalo Lab (part of Beken Corporation), which is designed for IoT applications.

Please note that prices and availability may change over time, and new RISC-V microcontrollers may have been introduced since my last update. To get the most accurate and up-to-date information on the cheapest RISC-V microcontrollers, it is recommended to check online retailers, electronics distributors, or manufacturers' websites.

Additionally, prices may vary depending on the quantity you wish to purchase, the specific model or configuration, and the region in which you are located.

---

do you know also ch32v003

Yes, the CH32V003 is a 32-bit ARM Cortex-M0+ microcontroller from WCH (Nanjing QinHeng Electronics). However, it is not a RISC-V microcontroller, but rather an ARM-based one. The CH32V003 comes with a variety of features and peripherals, making it suitable for various applications.

Key specifications of the CH32V003 microcontroller include:

Core: ARM Cortex-M0+ running at up to 48 MHz
Flash memory: Up to 64 KB
SRAM: Up to 8 KB
Peripherals: UART, SPI, I2C, GPIO, PWM, timers, ADC, DAC, and more
Power supply: 2.0 V to 3.6 V

It is essential to consult the manufacturer's website or product catalog for the most up-to-date information and the complete list of specifications and features for the CH32V003 microcontroller. Additionally, refer to the datasheet for detailed information on the microcontroller's functionality and programming.

is it available with 16kBflash?

Yes, the CH32V003 series from WCH (Nanjing QinHeng Electronics) does include microcontrollers with 16 KB of flash memory. The series offers different configurations with varying amounts of flash memory and SRAM, so you should be able to find a model that suits your specific needs.

However, it's essential to verify the availability and specifications of the exact model you require by consulting the manufacturer's website, product catalog, or datasheet. You may also want to check with electronics distributors and retailers to ensure the model with 16 KB of flash memory is available for purchase.

umm, according to this it is riscv https://github.com/openwch/ch32v003

Apologies for the confusion earlier. It seems that the CH32V003 is indeed a RISC-V microcontroller, based on the RV32IMAC core. The information provided in the GitHub repository you shared indicates that the CH32V003 series is part of the CH32V RISC-V microcontroller family from WCH (Nanjing QinHeng Electronics).

Thank you for providing this information. I apologize for any confusion caused by my previous response. Please refer to the GitHub repository, manufacturer's website, or product catalog for the most accurate and up-to-date information on the CH32V003 microcontroller and its features, including availability with 16 KB of flash memory.