

OpenBLT Bootloader - Reference Manual

Generated by Doxygen 1.9.4

| | |
|--|-----------|
| 1 OpenBLT Firmware Documentation | 1 |
| 1.1 Introduction | 1 |
| 1.2 Software Architecture | 1 |
| 1.3 Copyright and Licensing | 2 |
| 2 Module Index | 3 |
| 2.1 Modules | 3 |
| 3 Data Structure Index | 5 |
| 3.1 Data Structures | 5 |
| 4 File Index | 7 |
| 4.1 File List | 7 |
| 5 Module Documentation | 21 |
| 5.1 CAN driver of a port | 21 |
| 5.1.1 Detailed Description | 21 |
| 5.2 CPU driver of a port | 21 |
| 5.2.1 Detailed Description | 21 |
| 5.3 Flash driver of a port | 22 |
| 5.3.1 Detailed Description | 22 |
| 5.4 Compiler specifics of a port | 22 |
| 5.4.1 Detailed Description | 22 |
| 5.5 Modbus RTU driver of a port | 23 |
| 5.5.1 Detailed Description | 23 |
| 5.6 Non-volatile memory driver of a port | 23 |
| 5.6.1 Detailed Description | 23 |
| 5.7 RS232 UART driver of a port | 23 |
| 5.7.1 Detailed Description | 24 |
| 5.8 Target Port Template | 24 |
| 5.8.1 Detailed Description | 24 |
| 5.9 Timer driver of a port | 24 |
| 5.9.1 Detailed Description | 25 |
| 5.10 Type definitions of a port | 25 |
| 5.10.1 Detailed Description | 25 |
| 5.11 USB driver of a port | 25 |
| 5.11.1 Detailed Description | 25 |
| 5.12 Target ARMCM0 S32K11 | 26 |
| 5.12.1 Detailed Description | 26 |
| 5.13 Target ARMCM0 STM32C0 | 26 |
| 5.13.1 Detailed Description | 27 |
| 5.14 Target ARMCM0 STM32F0 | 27 |
| 5.14.1 Detailed Description | 28 |
| 5.15 Target ARMCM0 STM32G0 | 28 |

| | |
|-----------------------------|----|
| 5.15.1 Detailed Description | 29 |
| 5.16 Target ARMCM0 STM32L0 | 29 |
| 5.16.1 Detailed Description | 29 |
| 5.17 Target ARMCM0 XMC1 | 29 |
| 5.17.1 Detailed Description | 30 |
| 5.18 Target ARMCM33 STM32H5 | 30 |
| 5.18.1 Detailed Description | 31 |
| 5.19 Target ARMCM33 STM32L5 | 31 |
| 5.19.1 Detailed Description | 32 |
| 5.20 Target ARMCM33 STM32U5 | 32 |
| 5.20.1 Detailed Description | 33 |
| 5.21 Target ARMCM3 EFM32 | 33 |
| 5.21.1 Detailed Description | 33 |
| 5.22 Target ARMCM3 LM3S | 33 |
| 5.22.1 Detailed Description | 34 |
| 5.23 Target ARMCM3 STM32F1 | 34 |
| 5.23.1 Detailed Description | 35 |
| 5.24 Target ARMCM3 STM32F2 | 35 |
| 5.24.1 Detailed Description | 36 |
| 5.25 Target ARMCM3 STM32L1 | 36 |
| 5.25.1 Detailed Description | 36 |
| 5.26 Target ARMCM4 S32K14 | 37 |
| 5.26.1 Detailed Description | 37 |
| 5.27 Target ARMCM4 STM32F3 | 37 |
| 5.27.1 Detailed Description | 38 |
| 5.28 Target ARMCM4 STM32F4 | 38 |
| 5.28.1 Detailed Description | 39 |
| 5.29 Target ARMCM4 STM32G4 | 39 |
| 5.29.1 Detailed Description | 40 |
| 5.30 Target ARMCM4 STM32L4 | 40 |
| 5.30.1 Detailed Description | 41 |
| 5.31 Target ARMCM4 TM4C | 41 |
| 5.31.1 Detailed Description | 42 |
| 5.32 Target ARMCM4 XMC4 | 42 |
| 5.32.1 Detailed Description | 42 |
| 5.33 Target ARMCM7 STM32F7 | 43 |
| 5.33.1 Detailed Description | 43 |
| 5.34 Target ARMCM7 STM32H7 | 43 |
| 5.34.1 Detailed Description | 44 |
| 5.35 Bootloader Core | 44 |
| 5.35.1 Detailed Description | 46 |
| 5.36 Target HCS12 | 46 |

| | |
|--|-----------|
| 5.36.1 Detailed Description | 47 |
| 5.37 Bootloader Ports | 47 |
| 5.37.1 Detailed Description | 48 |
| 5.38 Target TRICORE TC2 | 48 |
| 5.38.1 Detailed Description | 49 |
| 5.39 Target TRICORE TC3 | 49 |
| 5.39.1 Detailed Description | 49 |
| 6 Data Structure Documentation | 51 |
| 6.1 tCanBusTiming Struct Reference | 51 |
| 6.1.1 Detailed Description | 51 |
| 6.1.2 Field Documentation | 51 |
| 6.1.2.1 phaseSeg1 | 51 |
| 6.1.2.2 phaseSeg2 | 52 |
| 6.1.2.3 propSeg | 52 |
| 6.1.2.4 timeQuanta | 52 |
| 6.1.2.5 tseg1 | 52 |
| 6.1.2.6 tseg2 | 53 |
| 6.2 tCanRegs Struct Reference | 53 |
| 6.2.1 Detailed Description | 54 |
| 6.2.2 Field Documentation | 54 |
| 6.2.2.1 cbtr0 | 54 |
| 6.2.2.2 cbtr1 | 55 |
| 6.2.2.3 cctl0 | 55 |
| 6.2.2.4 cctl1 | 55 |
| 6.2.2.5 cidac | 55 |
| 6.2.2.6 cidar0 | 55 |
| 6.2.2.7 cidar1 | 55 |
| 6.2.2.8 cidar2 | 55 |
| 6.2.2.9 cidar3 | 56 |
| 6.2.2.10 cidar4 | 56 |
| 6.2.2.11 cidar5 | 56 |
| 6.2.2.12 cidar6 | 56 |
| 6.2.2.13 cidar7 | 56 |
| 6.2.2.14 cidmr0 | 56 |
| 6.2.2.15 cidmr1 | 56 |
| 6.2.2.16 cidmr2 | 57 |
| 6.2.2.17 cidmr3 | 57 |
| 6.2.2.18 cidmr4 | 57 |
| 6.2.2.19 cidmr5 | 57 |
| 6.2.2.20 cidmr6 | 57 |
| 6.2.2.21 cidmr7 | 57 |

| | |
|--|----|
| 6.2.2.22 crflg | 57 |
| 6.2.2.23 crier | 58 |
| 6.2.2.24 crxerr | 58 |
| 6.2.2.25 ctaak | 58 |
| 6.2.2.26 ctarq | 58 |
| 6.2.2.27 ctbsel | 58 |
| 6.2.2.28 ctflg | 58 |
| 6.2.2.29 ctier | 58 |
| 6.2.2.30 ctxerr | 59 |
| 6.2.2.31 dummy1 | 59 |
| 6.2.2.32 rxSlot | 59 |
| 6.2.2.33 txSlot | 59 |
| 6.3 tCanRxMsgSlot Struct Reference | 59 |
| 6.3.1 Detailed Description | 59 |
| 6.3.2 Field Documentation | 60 |
| 6.3.2.1 dlr | 60 |
| 6.3.2.2 dsr | 60 |
| 6.3.2.3 dummy | 60 |
| 6.3.2.4 idr | 60 |
| 6.3.2.5 tstamp | 60 |
| 6.4 tCanTxMsgSlot Struct Reference | 60 |
| 6.4.1 Detailed Description | 61 |
| 6.4.2 Field Documentation | 61 |
| 6.4.2.1 dlr | 61 |
| 6.4.2.2 dsr | 61 |
| 6.4.2.3 idr | 61 |
| 6.4.2.4 tbpr | 61 |
| 6.4.2.5 tstamp | 62 |
| 6.5 tFatFsObjects Struct Reference | 62 |
| 6.5.1 Detailed Description | 62 |
| 6.5.2 Field Documentation | 62 |
| 6.5.2.1 file | 62 |
| 6.5.2.2 fs | 62 |
| 6.6 tFifoCtrl Struct Reference | 63 |
| 6.6.1 Detailed Description | 63 |
| 6.6.2 Field Documentation | 63 |
| 6.6.2.1 endptr | 63 |
| 6.6.2.2 entries | 63 |
| 6.6.2.3 fifoctrlptr | 64 |
| 6.6.2.4 handle | 64 |
| 6.6.2.5 length | 64 |
| 6.6.2.6 readptr | 64 |

| | |
|--|----|
| 6.6.2.7 startptr | 64 |
| 6.6.2.8 writeptr | 65 |
| 6.7 tFifoPipe Struct Reference | 65 |
| 6.7.1 Detailed Description | 65 |
| 6.7.2 Field Documentation | 65 |
| 6.7.2.1 data | 65 |
| 6.7.2.2 handle | 66 |
| 6.8 tFileEraseInfo Struct Reference | 66 |
| 6.8.1 Detailed Description | 66 |
| 6.8.2 Field Documentation | 66 |
| 6.8.2.1 start_address | 66 |
| 6.8.2.2 total_size | 67 |
| 6.9 tFlashBlockInfo Struct Reference | 67 |
| 6.9.1 Detailed Description | 67 |
| 6.10 tFlashPrescalerSysclockMapping Struct Reference | 68 |
| 6.10.1 Detailed Description | 68 |
| 6.10.2 Field Documentation | 68 |
| 6.10.2.1 prescaler | 68 |
| 6.10.2.2 sysclock_max | 68 |
| 6.10.2.3 sysclock_min | 68 |
| 6.11 tFlashRegs Struct Reference | 69 |
| 6.11.1 Detailed Description | 69 |
| 6.11.2 Field Documentation | 69 |
| 6.11.2.1 dfprot | 69 |
| 6.11.2.2 fccob | 69 |
| 6.11.2.3 fccobix | 70 |
| 6.11.2.4 fclkdiv | 70 |
| 6.11.2.5 fcmd | 70 |
| 6.11.2.6 fcngf | 70 |
| 6.11.2.7 fercfg | 70 |
| 6.11.2.8 ferstat | 70 |
| 6.11.2.9 fopt | 70 |
| 6.11.2.10 fprot | 71 |
| 6.11.2.11 frsv0 | 71 |
| 6.11.2.12 frsv1 | 71 |
| 6.11.2.13 frsv2 | 71 |
| 6.11.2.14 frsv3 | 71 |
| 6.11.2.15 frsv4 | 71 |
| 6.11.2.16 frsv5 | 72 |
| 6.11.2.17 frsv6 | 72 |
| 6.11.2.18 frsv7 | 72 |
| 6.11.2.19 fsec | 72 |

| | |
|--|----|
| 6.11.2.20 fstat | 72 |
| 6.11.2.21 ftstmod | 72 |
| 6.12 tFlashSector Struct Reference | 73 |
| 6.12.1 Detailed Description | 73 |
| 6.12.2 Field Documentation | 73 |
| 6.12.2.1 bank_num | 73 |
| 6.12.2.2 sector_num | 73 |
| 6.12.2.3 sector_size | 74 |
| 6.12.2.4 sector_start | 74 |
| 6.13 tSrecLineParseObject Struct Reference | 75 |
| 6.13.1 Detailed Description | 75 |
| 6.13.2 Field Documentation | 75 |
| 6.13.2.1 address | 75 |
| 6.13.2.2 data | 75 |
| 6.13.2.3 line | 75 |
| 6.14 tTimerRegs Struct Reference | 76 |
| 6.14.1 Detailed Description | 76 |
| 6.14.2 Field Documentation | 76 |
| 6.14.2.1 cforc | 76 |
| 6.14.2.2 oc7d | 76 |
| 6.14.2.3 oc7m | 77 |
| 6.14.2.4 tc | 77 |
| 6.14.2.5 tcnt | 77 |
| 6.14.2.6 tctl1 | 77 |
| 6.14.2.7 tctl2 | 77 |
| 6.14.2.8 tctl3 | 77 |
| 6.14.2.9 tctl4 | 77 |
| 6.14.2.10 tflg1 | 78 |
| 6.14.2.11 tflg2 | 78 |
| 6.14.2.12 tie | 78 |
| 6.14.2.13 tios | 78 |
| 6.14.2.14 tsqr1 | 78 |
| 6.14.2.15 tsqr2 | 78 |
| 6.14.2.16 ttov | 78 |
| 6.15 tXcpInfo Struct Reference | 79 |
| 6.15.1 Detailed Description | 79 |
| 6.15.2 Field Documentation | 79 |
| 6.15.2.1 connected | 79 |
| 6.15.2.2 ctoData | 79 |
| 6.15.2.3 ctoLen | 80 |
| 6.15.2.4 ctoPending | 80 |
| 6.15.2.5 mta | 80 |

| | |
|--|-----------|
| 6.15.2.6 protection | 80 |
| 6.15.2.7 s_n_k_resource | 81 |
| 6.16 uip_tcp_appstate_t Struct Reference | 81 |
| 6.16.1 Detailed Description | 81 |
| 7 File Documentation | 83 |
| 7.1 can.c File Reference | 83 |
| 7.1.1 Detailed Description | 84 |
| 7.1.2 Function Documentation | 84 |
| 7.1.2.1 CanGetSpeedConfig() | 84 |
| 7.1.2.2 CanInit() | 84 |
| 7.1.2.3 CanReceivePacket() | 85 |
| 7.1.2.4 CanTransmitPacket() | 85 |
| 7.1.3 Variable Documentation | 85 |
| 7.1.3.1 canTiming | 86 |
| 7.2 can.c File Reference | 86 |
| 7.2.1 Detailed Description | 87 |
| 7.2.2 Function Documentation | 88 |
| 7.2.2.1 CanDisabledModeEnter() | 88 |
| 7.2.2.2 CanDisabledModeExit() | 88 |
| 7.2.2.3 CanFreezeModeEnter() | 88 |
| 7.2.2.4 CanFreezeModeExit() | 89 |
| 7.2.2.5 CanGetSpeedConfig() | 89 |
| 7.2.2.6 CanInit() | 89 |
| 7.2.2.7 CanReceivePacket() | 90 |
| 7.2.2.8 CanTransmitPacket() | 90 |
| 7.2.3 Variable Documentation | 90 |
| 7.2.3.1 canTiming | 90 |
| 7.3 can.c File Reference | 91 |
| 7.3.1 Detailed Description | 92 |
| 7.3.2 Function Documentation | 92 |
| 7.3.2.1 CanGetSpeedConfig() | 92 |
| 7.3.2.2 CanInit() | 93 |
| 7.3.2.3 CanReceivePacket() | 93 |
| 7.3.2.4 CanTransmitPacket() | 94 |
| 7.3.3 Variable Documentation | 94 |
| 7.3.3.1 canTiming | 94 |
| 7.4 can.c File Reference | 95 |
| 7.4.1 Detailed Description | 96 |
| 7.4.2 Function Documentation | 96 |
| 7.4.2.1 CanGetSpeedConfig() | 96 |
| 7.4.2.2 CanInit() | 96 |

| | |
|------------------------------|-----|
| 7.4.2.3 CanReceivePacket() | 97 |
| 7.4.2.4 CanTransmitPacket() | 97 |
| 7.4.3 Variable Documentation | 97 |
| 7.4.3.1 canTiming | 97 |
| 7.5 can.c File Reference | 98 |
| 7.5.1 Detailed Description | 99 |
| 7.5.2 Function Documentation | 99 |
| 7.5.2.1 CanInit() | 99 |
| 7.5.2.2 CanReceivePacket() | 99 |
| 7.5.2.3 CanTransmitPacket() | 100 |
| 7.6 can.c File Reference | 100 |
| 7.6.1 Detailed Description | 101 |
| 7.6.2 Function Documentation | 101 |
| 7.6.2.1 CanGetSpeedConfig() | 101 |
| 7.6.2.2 CanInit() | 102 |
| 7.6.2.3 CanReceivePacket() | 102 |
| 7.6.2.4 CanTransmitPacket() | 103 |
| 7.6.3 Variable Documentation | 103 |
| 7.6.3.1 canTiming | 103 |
| 7.7 can.c File Reference | 104 |
| 7.7.1 Detailed Description | 105 |
| 7.7.2 Function Documentation | 105 |
| 7.7.2.1 CanGetSpeedConfig() | 105 |
| 7.7.2.2 CanInit() | 105 |
| 7.7.2.3 CanReceivePacket() | 106 |
| 7.7.2.4 CanTransmitPacket() | 106 |
| 7.7.3 Variable Documentation | 106 |
| 7.7.3.1 canTiming | 106 |
| 7.8 can.c File Reference | 107 |
| 7.8.1 Detailed Description | 108 |
| 7.8.2 Function Documentation | 108 |
| 7.8.2.1 CanGetSpeedConfig() | 108 |
| 7.8.2.2 CanInit() | 109 |
| 7.8.2.3 CanReceivePacket() | 109 |
| 7.8.2.4 CanTransmitPacket() | 109 |
| 7.8.3 Variable Documentation | 110 |
| 7.8.3.1 canTiming | 110 |
| 7.9 can.c File Reference | 111 |
| 7.9.1 Detailed Description | 112 |
| 7.9.2 Function Documentation | 112 |
| 7.9.2.1 CanInit() | 112 |
| 7.9.2.2 CanReceivePacket() | 112 |

| | |
|---------------------------------|-----|
| 7.9.2.3 CanSetBittiming() | 112 |
| 7.9.2.4 CanTransmitPacket() | 113 |
| 7.10 can.c File Reference | 113 |
| 7.10.1 Detailed Description | 114 |
| 7.10.2 Function Documentation | 114 |
| 7.10.2.1 CanGetSpeedConfig() | 114 |
| 7.10.2.2 CanInit() | 115 |
| 7.10.2.3 CanReceivePacket() | 115 |
| 7.10.2.4 CanTransmitPacket() | 116 |
| 7.10.3 Variable Documentation | 116 |
| 7.10.3.1 canTiming | 116 |
| 7.11 can.c File Reference | 117 |
| 7.11.1 Detailed Description | 118 |
| 7.11.2 Function Documentation | 118 |
| 7.11.2.1 CanGetSpeedConfig() | 118 |
| 7.11.2.2 CanInit() | 118 |
| 7.11.2.3 CanReceivePacket() | 119 |
| 7.11.2.4 CanTransmitPacket() | 119 |
| 7.11.3 Variable Documentation | 119 |
| 7.11.3.1 canTiming | 119 |
| 7.12 can.c File Reference | 120 |
| 7.12.1 Detailed Description | 121 |
| 7.12.2 Function Documentation | 122 |
| 7.12.2.1 CanDisabledModeEnter() | 122 |
| 7.12.2.2 CanDisabledModeExit() | 122 |
| 7.12.2.3 CanFreezeModeEnter() | 122 |
| 7.12.2.4 CanFreezeModeExit() | 123 |
| 7.12.2.5 CanGetSpeedConfig() | 123 |
| 7.12.2.6 CanInit() | 123 |
| 7.12.2.7 CanReceivePacket() | 124 |
| 7.12.2.8 CanTransmitPacket() | 124 |
| 7.12.3 Variable Documentation | 124 |
| 7.12.3.1 canTiming | 124 |
| 7.13 can.c File Reference | 125 |
| 7.13.1 Detailed Description | 126 |
| 7.13.2 Function Documentation | 126 |
| 7.13.2.1 CanGetSpeedConfig() | 126 |
| 7.13.2.2 CanInit() | 127 |
| 7.13.2.3 CanReceivePacket() | 127 |
| 7.13.2.4 CanTransmitPacket() | 128 |
| 7.13.3 Variable Documentation | 128 |
| 7.13.3.1 canTiming | 128 |

| | |
|---|-----|
| 7.14 can.c File Reference | 129 |
| 7.14.1 Detailed Description | 130 |
| 7.14.2 Function Documentation | 130 |
| 7.14.2.1 CanGetSpeedConfig() | 130 |
| 7.14.2.2 CanInit() | 130 |
| 7.14.2.3 CanReceivePacket() | 131 |
| 7.14.2.4 CanTransmitPacket() | 131 |
| 7.14.3 Variable Documentation | 131 |
| 7.14.3.1 canTiming | 131 |
| 7.15 can.c File Reference | 132 |
| 7.15.1 Detailed Description | 133 |
| 7.15.2 Function Documentation | 133 |
| 7.15.2.1 CanGetSpeedConfig() | 133 |
| 7.15.2.2 CanInit() | 134 |
| 7.15.2.3 CanReceivePacket() | 134 |
| 7.15.2.4 CanTransmitPacket() | 134 |
| 7.15.3 Variable Documentation | 135 |
| 7.15.3.1 canTiming | 135 |
| 7.16 can.c File Reference | 136 |
| 7.16.1 Detailed Description | 137 |
| 7.16.2 Function Documentation | 137 |
| 7.16.2.1 CanGetSpeedConfig() | 137 |
| 7.16.2.2 CanInit() | 137 |
| 7.16.2.3 CanReceivePacket() | 138 |
| 7.16.2.4 CanTransmitPacket() | 138 |
| 7.16.3 Variable Documentation | 138 |
| 7.16.3.1 canTiming | 138 |
| 7.17 can.c File Reference | 139 |
| 7.17.1 Detailed Description | 140 |
| 7.17.2 Function Documentation | 140 |
| 7.17.2.1 CanInit() | 140 |
| 7.17.2.2 CanReceivePacket() | 140 |
| 7.17.2.3 CanTransmitPacket() | 141 |
| 7.18 can.c File Reference | 141 |
| 7.18.1 Detailed Description | 142 |
| 7.18.2 Function Documentation | 142 |
| 7.18.2.1 CanGetSpeedConfig() | 142 |
| 7.18.2.2 CanInit() | 143 |
| 7.18.2.3 CanReceivePacket() | 143 |
| 7.18.2.4 CanTransmitPacket() | 144 |
| 7.18.3 Variable Documentation | 144 |
| 7.18.3.1 canTiming | 144 |

| | |
|---|-----|
| 7.19 can.c File Reference | 145 |
| 7.19.1 Detailed Description | 146 |
| 7.19.2 Function Documentation | 146 |
| 7.19.2.1 CanGetSpeedConfig() | 146 |
| 7.19.2.2 CanInit() | 146 |
| 7.19.2.3 CanReceivePacket() | 147 |
| 7.19.2.4 CanTransmitPacket() | 147 |
| 7.19.3 Variable Documentation | 147 |
| 7.19.3.1 canTiming | 147 |
| 7.20 can.c File Reference | 148 |
| 7.20.1 Detailed Description | 149 |
| 7.20.2 Function Documentation | 150 |
| 7.20.2.1 CanGetSpeedConfig() | 150 |
| 7.20.2.2 CanInit() | 150 |
| 7.20.2.3 CanReceivePacket() | 150 |
| 7.20.2.4 CanTransmitPacket() | 151 |
| 7.20.3 Variable Documentation | 151 |
| 7.20.3.1 canTiming | 151 |
| 7.21 can.c File Reference | 152 |
| 7.21.1 Detailed Description | 153 |
| 7.21.2 Function Documentation | 153 |
| 7.21.2.1 CanGetSpeedConfig() | 153 |
| 7.21.2.2 CanInit() | 154 |
| 7.21.2.3 CanReceivePacket() | 154 |
| 7.21.2.4 CanTransmitPacket() | 154 |
| 7.21.3 Variable Documentation | 155 |
| 7.21.3.1 canTiming | 155 |
| 7.22 can.c File Reference | 156 |
| 7.22.1 Detailed Description | 157 |
| 7.22.2 Function Documentation | 157 |
| 7.22.2.1 CanGetSpeedConfig() | 157 |
| 7.22.2.2 CanInit() | 158 |
| 7.22.2.3 CanReceivePacket() | 158 |
| 7.22.2.4 CanTransmitPacket() | 158 |
| 7.22.3 Variable Documentation | 159 |
| 7.22.3.1 canTiming | 159 |
| 7.23 cpu.c File Reference | 159 |
| 7.23.1 Detailed Description | 160 |
| 7.23.2 Function Documentation | 160 |
| 7.23.2.1 CpuInit() | 160 |
| 7.23.2.2 CpuMemcpy() | 160 |
| 7.23.2.3 CpuMemSet() | 161 |

| | |
|--------------------------------|-----|
| 7.23.2.4 CpuStartUserProgram() | 161 |
| 7.24 cpu.c File Reference | 162 |
| 7.24.1 Detailed Description | 162 |
| 7.24.2 Function Documentation | 162 |
| 7.24.2.1 CpuInit() | 163 |
| 7.24.2.2 CpuMemCopy() | 163 |
| 7.24.2.3 CpuMemSet() | 163 |
| 7.24.2.4 CpuStartUserProgram() | 164 |
| 7.25 cpu.c File Reference | 164 |
| 7.25.1 Detailed Description | 165 |
| 7.25.2 Function Documentation | 165 |
| 7.25.2.1 CpuInit() | 165 |
| 7.25.2.2 CpuMemCopy() | 165 |
| 7.25.2.3 CpuMemSet() | 166 |
| 7.25.2.4 CpuStartUserProgram() | 166 |
| 7.26 cpu.c File Reference | 166 |
| 7.26.1 Detailed Description | 167 |
| 7.26.2 Function Documentation | 167 |
| 7.26.2.1 CpuInit() | 168 |
| 7.26.2.2 CpuMemCopy() | 168 |
| 7.26.2.3 CpuMemSet() | 168 |
| 7.26.2.4 CpuStartUserProgram() | 169 |
| 7.27 cpu.c File Reference | 169 |
| 7.27.1 Detailed Description | 170 |
| 7.27.2 Function Documentation | 170 |
| 7.27.2.1 CpuInit() | 170 |
| 7.27.2.2 CpuMemCopy() | 170 |
| 7.27.2.3 CpuMemSet() | 171 |
| 7.27.2.4 CpuStartUserProgram() | 171 |
| 7.28 cpu.c File Reference | 171 |
| 7.28.1 Detailed Description | 172 |
| 7.28.2 Function Documentation | 172 |
| 7.28.2.1 CpuInit() | 172 |
| 7.28.2.2 CpuMemCopy() | 172 |
| 7.28.2.3 CpuMemSet() | 173 |
| 7.28.2.4 CpuStartUserProgram() | 173 |
| 7.29 cpu.c File Reference | 174 |
| 7.29.1 Detailed Description | 174 |
| 7.29.2 Function Documentation | 174 |
| 7.29.2.1 CpuInit() | 175 |
| 7.29.2.2 CpuMemCopy() | 175 |
| 7.29.2.3 CpuMemSet() | 175 |

| | |
|--------------------------------|-----|
| 7.29.2.4 CpuStartUserProgram() | 176 |
| 7.30 cpu.c File Reference | 176 |
| 7.30.1 Detailed Description | 177 |
| 7.30.2 Function Documentation | 177 |
| 7.30.2.1 CpuInit() | 177 |
| 7.30.2.2 CpuMemCopy() | 177 |
| 7.30.2.3 CpuMemSet() | 178 |
| 7.30.2.4 CpuStartUserProgram() | 178 |
| 7.31 cpu.c File Reference | 178 |
| 7.31.1 Detailed Description | 179 |
| 7.31.2 Function Documentation | 179 |
| 7.31.2.1 CpuInit() | 179 |
| 7.31.2.2 CpuMemCopy() | 179 |
| 7.31.2.3 CpuMemSet() | 180 |
| 7.31.2.4 CpuStartUserProgram() | 180 |
| 7.32 cpu.c File Reference | 181 |
| 7.32.1 Detailed Description | 181 |
| 7.32.2 Function Documentation | 181 |
| 7.32.2.1 CpuInit() | 182 |
| 7.32.2.2 CpuMemCopy() | 182 |
| 7.32.2.3 CpuMemSet() | 182 |
| 7.32.2.4 CpuStartUserProgram() | 183 |
| 7.33 cpu.c File Reference | 183 |
| 7.33.1 Detailed Description | 184 |
| 7.33.2 Function Documentation | 184 |
| 7.33.2.1 CpuInit() | 184 |
| 7.33.2.2 CpuMemCopy() | 184 |
| 7.33.2.3 CpuMemSet() | 185 |
| 7.33.2.4 CpuStartUserProgram() | 185 |
| 7.34 cpu.c File Reference | 185 |
| 7.34.1 Detailed Description | 186 |
| 7.34.2 Function Documentation | 186 |
| 7.34.2.1 CpuInit() | 186 |
| 7.34.2.2 CpuMemCopy() | 186 |
| 7.34.2.3 CpuMemSet() | 187 |
| 7.34.2.4 CpuStartUserProgram() | 187 |
| 7.35 cpu.c File Reference | 188 |
| 7.35.1 Detailed Description | 188 |
| 7.35.2 Function Documentation | 188 |
| 7.35.2.1 CpuInit() | 189 |
| 7.35.2.2 CpuMemCopy() | 189 |
| 7.35.2.3 CpuMemSet() | 189 |

| | |
|--------------------------------|-----|
| 7.35.2.4 CpuStartUserProgram() | 190 |
| 7.36 cpu.c File Reference | 190 |
| 7.36.1 Detailed Description | 191 |
| 7.36.2 Function Documentation | 191 |
| 7.36.2.1 CpuInit() | 191 |
| 7.36.2.2 CpuMemCopy() | 191 |
| 7.36.2.3 CpuMemSet() | 191 |
| 7.36.2.4 CpuStartUserProgram() | 192 |
| 7.37 cpu.c File Reference | 192 |
| 7.37.1 Detailed Description | 193 |
| 7.37.2 Function Documentation | 193 |
| 7.37.2.1 CpuInit() | 193 |
| 7.37.2.2 CpuMemCopy() | 193 |
| 7.37.2.3 CpuMemSet() | 194 |
| 7.37.2.4 CpuStartUserProgram() | 194 |
| 7.38 cpu.c File Reference | 195 |
| 7.38.1 Detailed Description | 195 |
| 7.38.2 Function Documentation | 195 |
| 7.38.2.1 CpuInit() | 196 |
| 7.38.2.2 CpuMemCopy() | 196 |
| 7.38.2.3 CpuMemSet() | 196 |
| 7.38.2.4 CpuStartUserProgram() | 197 |
| 7.39 cpu.c File Reference | 197 |
| 7.39.1 Detailed Description | 198 |
| 7.39.2 Function Documentation | 198 |
| 7.39.2.1 CpuInit() | 198 |
| 7.39.2.2 CpuMemCopy() | 198 |
| 7.39.2.3 CpuMemSet() | 198 |
| 7.39.2.4 CpuStartUserProgram() | 199 |
| 7.40 cpu.c File Reference | 199 |
| 7.40.1 Detailed Description | 200 |
| 7.40.2 Function Documentation | 200 |
| 7.40.2.1 CpuInit() | 200 |
| 7.40.2.2 CpuMemCopy() | 200 |
| 7.40.2.3 CpuMemSet() | 201 |
| 7.40.2.4 CpuStartUserProgram() | 201 |
| 7.41 cpu.c File Reference | 202 |
| 7.41.1 Detailed Description | 202 |
| 7.41.2 Function Documentation | 202 |
| 7.41.2.1 CpuInit() | 203 |
| 7.41.2.2 CpuMemCopy() | 203 |
| 7.41.2.3 CpuMemSet() | 203 |

| | |
|---|-----|
| 7.41.2.4 CpuStartUserProgram() | 204 |
| 7.42 cpu.c File Reference | 204 |
| 7.42.1 Detailed Description | 205 |
| 7.42.2 Function Documentation | 205 |
| 7.42.2.1 Cpulnit() | 205 |
| 7.42.2.2 CpuMemCopy() | 205 |
| 7.42.2.3 CpuMemSet() | 205 |
| 7.42.2.4 CpuStartUserProgram() | 206 |
| 7.43 cpu.c File Reference | 206 |
| 7.43.1 Detailed Description | 207 |
| 7.43.2 Function Documentation | 207 |
| 7.43.2.1 Cpulnit() | 207 |
| 7.43.2.2 CpuMemCopy() | 207 |
| 7.43.2.3 CpuMemSet() | 208 |
| 7.43.2.4 CpuStartUserProgram() | 208 |
| 7.44 cpu.c File Reference | 209 |
| 7.44.1 Detailed Description | 209 |
| 7.44.2 Function Documentation | 209 |
| 7.44.2.1 Cpulnit() | 210 |
| 7.44.2.2 CpuMemCopy() | 210 |
| 7.44.2.3 CpuMemSet() | 210 |
| 7.44.2.4 CpuStartUserProgram() | 211 |
| 7.45 cpu.c File Reference | 211 |
| 7.45.1 Detailed Description | 212 |
| 7.45.2 Function Documentation | 212 |
| 7.45.2.1 Cpulnit() | 212 |
| 7.45.2.2 CpuMemCopy() | 212 |
| 7.45.2.3 CpuMemSet() | 212 |
| 7.45.2.4 CpuStartUserProgram() | 213 |
| 7.46 cpu.c File Reference | 213 |
| 7.46.1 Detailed Description | 214 |
| 7.46.2 Function Documentation | 214 |
| 7.46.2.1 Cpulnit() | 214 |
| 7.46.2.2 CpuMemCopy() | 214 |
| 7.46.2.3 CpuMemSet() | 215 |
| 7.46.2.4 CpuStartUserProgram() | 215 |
| 7.47 cpu.c File Reference | 216 |
| 7.47.1 Detailed Description | 216 |
| 7.47.2 Macro Definition Documentation | 216 |
| 7.47.2.1 CPU_USER_PROGRAM_STARTADDR_PTR | 217 |
| 7.47.3 Function Documentation | 217 |
| 7.47.3.1 Cpulnit() | 217 |

| | |
|---|-----|
| 7.47.3.2 CpuMemCopy() | 217 |
| 7.47.3.3 CpuMemSet() | 218 |
| 7.47.3.4 CpuStartUserProgram() | 218 |
| 7.48 cpu.c File Reference | 218 |
| 7.48.1 Detailed Description | 219 |
| 7.48.2 Function Documentation | 219 |
| 7.48.2.1 CpuEnableUncorrectableBitErrorTrap() | 219 |
| 7.48.2.2 CpuInit() | 220 |
| 7.48.2.3 CpuMemCopy() | 220 |
| 7.48.2.4 CpuMemSet() | 220 |
| 7.48.2.5 CpuStartUserProgram() | 221 |
| 7.49 cpu.c File Reference | 221 |
| 7.49.1 Detailed Description | 222 |
| 7.49.2 Function Documentation | 222 |
| 7.49.2.1 CpuEnableUncorrectableEccErrorTrap() | 222 |
| 7.49.2.2 CpuInit() | 222 |
| 7.49.2.3 CpuMemCopy() | 222 |
| 7.49.2.4 CpuMemSet() | 223 |
| 7.49.2.5 CpuStartUserProgram() | 223 |
| 7.50 flash.c File Reference | 224 |
| 7.50.1 Detailed Description | 225 |
| 7.50.2 Function Documentation | 226 |
| 7.50.2.1 FlashAddToBlock() | 226 |
| 7.50.2.2 FlashDone() | 226 |
| 7.50.2.3 FlashErase() | 226 |
| 7.50.2.4 FlashEraseSectors() | 227 |
| 7.50.2.5 FlashGetSectorIdx() | 227 |
| 7.50.2.6 Flash GetUserProgBaseAddress() | 228 |
| 7.50.2.7 FlashInit() | 228 |
| 7.50.2.8 FlashInitBlock() | 228 |
| 7.50.2.9 FlashSwitchBlock() | 229 |
| 7.50.2.10 FlashVerifyChecksum() | 229 |
| 7.50.2.11 FlashWrite() | 229 |
| 7.50.2.12 FlashWriteBlock() | 230 |
| 7.50.2.13 FlashWriteChecksum() | 230 |
| 7.50.3 Variable Documentation | 230 |
| 7.50.3.1 blockInfo | 231 |
| 7.50.3.2 bootBlockInfo | 231 |
| 7.50.3.3 flashLayout | 231 |
| 7.51 flash.c File Reference | 232 |
| 7.51.1 Detailed Description | 234 |
| 7.51.2 Function Documentation | 234 |

| | |
|---|-----|
| 7.51.2.1 FlashAddToBlock() | 234 |
| 7.51.2.2 FlashCommandSequence() | 234 |
| 7.51.2.3 FlashDone() | 235 |
| 7.51.2.4 FlashErase() | 235 |
| 7.51.2.5 FlashEraseSectors() | 236 |
| 7.51.2.6 FlashGetSectorIdx() | 236 |
| 7.51.2.7 Flash GetUserProgBaseAddress() | 236 |
| 7.51.2.8 FlashInit() | 237 |
| 7.51.2.9 FlashInitBlock() | 237 |
| 7.51.2.10 FlashSwitchBlock() | 237 |
| 7.51.2.11 FlashVerifyChecksum() | 238 |
| 7.51.2.12 FlashWrite() | 238 |
| 7.51.2.13 FlashWriteBlock() | 239 |
| 7.51.2.14 FlashWriteChecksum() | 239 |
| 7.51.3 Variable Documentation | 239 |
| 7.51.3.1 blockInfo | 239 |
| 7.51.3.2 bootBlockInfo | 240 |
| 7.52 flash.c File Reference | 240 |
| 7.52.1 Detailed Description | 242 |
| 7.52.2 Function Documentation | 242 |
| 7.52.2.1 FlashAddToBlock() | 242 |
| 7.52.2.2 FlashDone() | 243 |
| 7.52.2.3 FlashErase() | 243 |
| 7.52.2.4 FlashEraseSectors() | 243 |
| 7.52.2.5 FlashGetPage() | 244 |
| 7.52.2.6 FlashGetSector() | 244 |
| 7.52.2.7 FlashGetSectorBaseAddr() | 245 |
| 7.52.2.8 FlashGetSectorSize() | 245 |
| 7.52.2.9 Flash GetUserProgBaseAddress() | 245 |
| 7.52.2.10 FlashInit() | 246 |
| 7.52.2.11 FlashInitBlock() | 246 |
| 7.52.2.12 FlashSwitchBlock() | 246 |
| 7.52.2.13 FlashVerifyChecksum() | 247 |
| 7.52.2.14 FlashWrite() | 247 |
| 7.52.2.15 FlashWriteBlock() | 247 |
| 7.52.2.16 FlashWriteChecksum() | 248 |
| 7.52.3 Variable Documentation | 248 |
| 7.52.3.1 blockInfo | 248 |
| 7.52.3.2 bootBlockInfo | 249 |
| 7.52.3.3 flashLayout | 249 |
| 7.53 flash.c File Reference | 249 |
| 7.53.1 Detailed Description | 251 |

| | |
|--|-----|
| 7.53.2 Function Documentation | 251 |
| 7.53.2.1 FlashAddToBlock() | 251 |
| 7.53.2.2 FlashDone() | 252 |
| 7.53.2.3 FlashErase() | 252 |
| 7.53.2.4 FlashEraseSectors() | 253 |
| 7.53.2.5 FlashGetSector() | 253 |
| 7.53.2.6 FlashGetSectorBaseAddr() | 253 |
| 7.53.2.7 FlashGetSectorSize() | 254 |
| 7.53.2.8 Flash GetUserProgBaseAddress() | 254 |
| 7.53.2.9 FlashInit() | 254 |
| 7.53.2.10 FlashInitBlock() | 255 |
| 7.53.2.11 FlashSwitchBlock() | 255 |
| 7.53.2.12 FlashVerifyChecksum() | 255 |
| 7.53.2.13 FlashWrite() | 256 |
| 7.53.2.14 FlashWriteBlock() | 256 |
| 7.53.2.15 FlashWriteChecksum() | 256 |
| 7.53.3 Variable Documentation | 257 |
| 7.53.3.1 blockInfo | 257 |
| 7.53.3.2 bootBlockInfo | 257 |
| 7.53.3.3 flashLayout | 258 |
| 7.54 flash.c File Reference | 258 |
| 7.54.1 Detailed Description | 260 |
| 7.54.2 Function Documentation | 260 |
| 7.54.2.1 FlashAddToBlock() | 260 |
| 7.54.2.2 FlashDone() | 261 |
| 7.54.2.3 FlashErase() | 261 |
| 7.54.2.4 FlashEraseSectors() | 261 |
| 7.54.2.5 FlashGetBank() | 262 |
| 7.54.2.6 FlashGetPage() | 262 |
| 7.54.2.7 FlashGetSector() | 263 |
| 7.54.2.8 FlashGetSectorBaseAddr() | 263 |
| 7.54.2.9 FlashGetSectorSize() | 263 |
| 7.54.2.10 Flash GetUserProgBaseAddress() | 264 |
| 7.54.2.11 FlashInit() | 264 |
| 7.54.2.12 FlashInitBlock() | 264 |
| 7.54.2.13 FlashSwitchBlock() | 265 |
| 7.54.2.14 FlashVerifyChecksum() | 265 |
| 7.54.2.15 FlashWrite() | 266 |
| 7.54.2.16 FlashWriteBlock() | 267 |
| 7.54.2.17 FlashWriteChecksum() | 267 |
| 7.54.3 Variable Documentation | 267 |
| 7.54.3.1 blockInfo | 268 |

| | |
|---|-----|
| 7.54.3.2 bootBlockInfo | 268 |
| 7.54.3.3 flashLayout | 268 |
| 7.55 flash.c File Reference | 269 |
| 7.55.1 Detailed Description | 270 |
| 7.55.2 Function Documentation | 270 |
| 7.55.2.1 FlashAddToBlock() | 271 |
| 7.55.2.2 FlashDone() | 271 |
| 7.55.2.3 FlashErase() | 271 |
| 7.55.2.4 Flash GetUserProgBaseAddress() | 272 |
| 7.55.2.5 FlashInit() | 272 |
| 7.55.2.6 FlashInitBlock() | 272 |
| 7.55.2.7 FlashSwitchBlock() | 273 |
| 7.55.2.8 FlashVerifyChecksum() | 273 |
| 7.55.2.9 FlashWrite() | 274 |
| 7.55.2.10 FlashWriteBlock() | 275 |
| 7.55.2.11 FlashWriteChecksum() | 275 |
| 7.55.3 Variable Documentation | 275 |
| 7.55.3.1 blockInfo | 276 |
| 7.55.3.2 bootBlockInfo | 276 |
| 7.55.3.3 flashLayout | 276 |
| 7.56 flash.c File Reference | 277 |
| 7.56.1 Detailed Description | 278 |
| 7.56.2 Function Documentation | 279 |
| 7.56.2.1 FlashAddToBlock() | 279 |
| 7.56.2.2 FlashDone() | 279 |
| 7.56.2.3 FlashErase() | 279 |
| 7.56.2.4 FlashEraseSectors() | 280 |
| 7.56.2.5 FlashGetSector() | 280 |
| 7.56.2.6 FlashGetSectorBaseAddr() | 281 |
| 7.56.2.7 Flash GetUserProgBaseAddress() | 281 |
| 7.56.2.8 FlashInit() | 281 |
| 7.56.2.9 FlashInitBlock() | 281 |
| 7.56.2.10 FlashSwitchBlock() | 282 |
| 7.56.2.11 FlashVerifyChecksum() | 282 |
| 7.56.2.12 FlashWrite() | 283 |
| 7.56.2.13 FlashWriteBlock() | 284 |
| 7.56.2.14 FlashWriteChecksum() | 284 |
| 7.56.3 Variable Documentation | 284 |
| 7.56.3.1 blockInfo | 285 |
| 7.56.3.2 bootBlockInfo | 285 |
| 7.56.3.3 flashLayout | 285 |
| 7.57 flash.c File Reference | 286 |

| | |
|---|-----|
| 7.57.1 Detailed Description | 288 |
| 7.57.2 Function Documentation | 288 |
| 7.57.2.1 FlashAddToBlock() | 288 |
| 7.57.2.2 FlashDone() | 288 |
| 7.57.2.3 FlashEmptyCheckSector() | 289 |
| 7.57.2.4 FlashErase() | 290 |
| 7.57.2.5 FlashEraseSectors() | 290 |
| 7.57.2.6 FlashGetBank() | 291 |
| 7.57.2.7 FlashGetPage() | 291 |
| 7.57.2.8 FlashGetSectorIdx() | 291 |
| 7.57.2.9 Flash GetUserProgBaseAddress() | 292 |
| 7.57.2.10 FlashInit() | 292 |
| 7.57.2.11 FlashInitBlock() | 292 |
| 7.57.2.12 FlashSwitchBlock() | 293 |
| 7.57.2.13 FlashVerifyChecksum() | 293 |
| 7.57.2.14 FlashWrite() | 294 |
| 7.57.2.15 FlashWriteBlock() | 295 |
| 7.57.2.16 FlashWriteChecksum() | 295 |
| 7.57.3 Variable Documentation | 295 |
| 7.57.3.1 blockInfo | 296 |
| 7.57.3.2 bootBlockInfo | 296 |
| 7.57.3.3 flashLayout | 296 |
| 7.58 flash.c File Reference | 297 |
| 7.58.1 Detailed Description | 299 |
| 7.58.2 Function Documentation | 299 |
| 7.58.2.1 FlashAddToBlock() | 299 |
| 7.58.2.2 FlashDone() | 299 |
| 7.58.2.3 FlashErase() | 300 |
| 7.58.2.4 FlashEraseSectors() | 300 |
| 7.58.2.5 FlashGetBank() | 300 |
| 7.58.2.6 FlashGetPage() | 301 |
| 7.58.2.7 FlashGetPageSize() | 301 |
| 7.58.2.8 FlashGetSectorIdx() | 301 |
| 7.58.2.9 Flash GetUserProgBaseAddress() | 302 |
| 7.58.2.10 FlashInit() | 302 |
| 7.58.2.11 FlashInitBlock() | 302 |
| 7.58.2.12 FlashIsDualBankMode() | 303 |
| 7.58.2.13 FlashSwitchBlock() | 303 |
| 7.58.2.14 FlashVerifyChecksum() | 304 |
| 7.58.2.15 FlashWrite() | 304 |
| 7.58.2.16 FlashWriteBlock() | 304 |
| 7.58.2.17 FlashWriteChecksum() | 305 |

| | |
|---|-----|
| 7.58.3 Variable Documentation | 305 |
| 7.58.3.1 blockInfo | 305 |
| 7.58.3.2 bootBlockInfo | 305 |
| 7.58.3.3 flashLayout | 306 |
| 7.59 flash.c File Reference | 306 |
| 7.59.1 Detailed Description | 308 |
| 7.59.2 Function Documentation | 308 |
| 7.59.2.1 FlashAddToBlock() | 308 |
| 7.59.2.2 FlashDone() | 309 |
| 7.59.2.3 FlashEmptyCheckSector() | 309 |
| 7.59.2.4 FlashErase() | 309 |
| 7.59.2.5 FlashEraseSectors() | 310 |
| 7.59.2.6 FlashGetBank() | 310 |
| 7.59.2.7 FlashGetPage() | 311 |
| 7.59.2.8 FlashGetSectorIdx() | 311 |
| 7.59.2.9 Flash GetUserProgBaseAddress() | 311 |
| 7.59.2.10 FlashInit() | 312 |
| 7.59.2.11 FlashInitBlock() | 312 |
| 7.59.2.12 FlashSwitchBlock() | 312 |
| 7.59.2.13 FlashVerifyChecksum() | 313 |
| 7.59.2.14 FlashWrite() | 313 |
| 7.59.2.15 FlashWriteBlock() | 313 |
| 7.59.2.16 FlashWriteChecksum() | 314 |
| 7.59.3 Variable Documentation | 314 |
| 7.59.3.1 blockInfo | 314 |
| 7.59.3.2 bootBlockInfo | 315 |
| 7.59.3.3 flashLayout | 315 |
| 7.60 flash.c File Reference | 315 |
| 7.60.1 Detailed Description | 317 |
| 7.60.2 Function Documentation | 317 |
| 7.60.2.1 FlashAddToBlock() | 317 |
| 7.60.2.2 FlashCalcPageSize() | 318 |
| 7.60.2.3 FlashDone() | 318 |
| 7.60.2.4 FlashErase() | 318 |
| 7.60.2.5 FlashEraseSectors() | 319 |
| 7.60.2.6 FlashGetSector() | 319 |
| 7.60.2.7 FlashGetSectorBaseAddr() | 320 |
| 7.60.2.8 FlashGetSectorSize() | 320 |
| 7.60.2.9 Flash GetUserProgBaseAddress() | 320 |
| 7.60.2.10 FlashInit() | 321 |
| 7.60.2.11 FlashInitBlock() | 321 |
| 7.60.2.12 FlashSwitchBlock() | 321 |

| | |
|---|-----|
| 7.60.2.13 FlashVerifyChecksum() | 322 |
| 7.60.2.14 FlashWrite() | 322 |
| 7.60.2.15 FlashWriteBlock() | 322 |
| 7.60.2.16 FlashWriteChecksum() | 323 |
| 7.60.3 Variable Documentation | 323 |
| 7.60.3.1 blockInfo | 323 |
| 7.60.3.2 bootBlockInfo | 324 |
| 7.60.3.3 flashLayout | 324 |
| 7.61 flash.c File Reference | 324 |
| 7.61.1 Detailed Description | 326 |
| 7.61.2 Function Documentation | 326 |
| 7.61.2.1 FlashAddToBlock() | 326 |
| 7.61.2.2 FlashDone() | 327 |
| 7.61.2.3 FlashErase() | 327 |
| 7.61.2.4 FlashEraseSectors() | 328 |
| 7.61.2.5 FlashGetSector() | 328 |
| 7.61.2.6 FlashGetSectorBaseAddr() | 328 |
| 7.61.2.7 FlashGetSectorSize() | 329 |
| 7.61.2.8 Flash GetUserProgBaseAddress() | 329 |
| 7.61.2.9 FlashInit() | 329 |
| 7.61.2.10 FlashInitBlock() | 330 |
| 7.61.2.11 FlashSwitchBlock() | 330 |
| 7.61.2.12 FlashVerifyChecksum() | 330 |
| 7.61.2.13 FlashWrite() | 331 |
| 7.61.2.14 FlashWriteBlock() | 331 |
| 7.61.2.15 FlashWriteChecksum() | 331 |
| 7.61.3 Variable Documentation | 332 |
| 7.61.3.1 blockInfo | 332 |
| 7.61.3.2 bootBlockInfo | 332 |
| 7.61.3.3 flashLayout | 333 |
| 7.62 flash.c File Reference | 333 |
| 7.62.1 Detailed Description | 335 |
| 7.62.2 Function Documentation | 335 |
| 7.62.2.1 FlashAddToBlock() | 335 |
| 7.62.2.2 FlashDone() | 336 |
| 7.62.2.3 FlashErase() | 336 |
| 7.62.2.4 Flash GetUserProgBaseAddress() | 336 |
| 7.62.2.5 FlashInit() | 337 |
| 7.62.2.6 FlashInitBlock() | 337 |
| 7.62.2.7 FlashSwitchBlock() | 337 |
| 7.62.2.8 FlashVerifyChecksum() | 338 |
| 7.62.2.9 FlashWrite() | 338 |

| | |
|---|-----|
| 7.62.2.10 FlashWriteBlock() | 338 |
| 7.62.2.11 FlashWriteChecksum() | 339 |
| 7.62.3 Variable Documentation | 339 |
| 7.62.3.1 blockInfo | 339 |
| 7.62.3.2 bootBlockInfo | 340 |
| 7.62.3.3 flashLayout | 340 |
| 7.63 flash.c File Reference | 340 |
| 7.63.1 Detailed Description | 342 |
| 7.63.2 Function Documentation | 342 |
| 7.63.2.1 FlashAddToBlock() | 342 |
| 7.63.2.2 FlashDone() | 343 |
| 7.63.2.3 FlashEmptyCheckSector() | 343 |
| 7.63.2.4 FlashErase() | 343 |
| 7.63.2.5 FlashEraseSectors() | 344 |
| 7.63.2.6 FlashGetSector() | 344 |
| 7.63.2.7 Flash GetUserProgBaseAddress() | 345 |
| 7.63.2.8 FlashInit() | 345 |
| 7.63.2.9 FlashInitBlock() | 345 |
| 7.63.2.10 FlashSwitchBlock() | 346 |
| 7.63.2.11 FlashVerifyChecksum() | 346 |
| 7.63.2.12 FlashWrite() | 346 |
| 7.63.2.13 FlashWriteBlock() | 347 |
| 7.63.2.14 FlashWriteChecksum() | 347 |
| 7.63.3 Variable Documentation | 347 |
| 7.63.3.1 blockInfo | 348 |
| 7.63.3.2 bootBlockInfo | 348 |
| 7.63.3.3 flashLayout | 348 |
| 7.64 flash.c File Reference | 349 |
| 7.64.1 Detailed Description | 350 |
| 7.64.2 Function Documentation | 350 |
| 7.64.2.1 FlashAddToBlock() | 351 |
| 7.64.2.2 FlashDone() | 351 |
| 7.64.2.3 FlashErase() | 351 |
| 7.64.2.4 Flash GetUserProgBaseAddress() | 352 |
| 7.64.2.5 FlashInit() | 352 |
| 7.64.2.6 FlashInitBlock() | 352 |
| 7.64.2.7 FlashSwitchBlock() | 353 |
| 7.64.2.8 FlashVerifyChecksum() | 353 |
| 7.64.2.9 FlashWrite() | 354 |
| 7.64.2.10 FlashWriteBlock() | 355 |
| 7.64.2.11 FlashWriteChecksum() | 355 |
| 7.64.3 Variable Documentation | 355 |

| | |
|---|-----|
| 7.64.3.1 blockInfo | 356 |
| 7.64.3.2 bootBlockInfo | 356 |
| 7.64.3.3 flashLayout | 356 |
| 7.65 flash.c File Reference | 357 |
| 7.65.1 Detailed Description | 358 |
| 7.65.2 Function Documentation | 359 |
| 7.65.2.1 FlashAddToBlock() | 359 |
| 7.65.2.2 FlashCommandSequence() | 359 |
| 7.65.2.3 FlashDone() | 360 |
| 7.65.2.4 FlashErase() | 360 |
| 7.65.2.5 FlashEraseSectors() | 360 |
| 7.65.2.6 FlashGetSectorIdx() | 361 |
| 7.65.2.7 Flash GetUserProgBaseAddress() | 361 |
| 7.65.2.8 FlashInit() | 362 |
| 7.65.2.9 FlashInitBlock() | 362 |
| 7.65.2.10 FlashSwitchBlock() | 362 |
| 7.65.2.11 FlashVerifyChecksum() | 363 |
| 7.65.2.12 FlashWrite() | 363 |
| 7.65.2.13 FlashWriteBlock() | 363 |
| 7.65.2.14 FlashWriteChecksum() | 364 |
| 7.65.3 Variable Documentation | 364 |
| 7.65.3.1 blockInfo | 364 |
| 7.65.3.2 bootBlockInfo | 365 |
| 7.66 flash.c File Reference | 365 |
| 7.66.1 Detailed Description | 367 |
| 7.66.2 Function Documentation | 367 |
| 7.66.2.1 FlashAddToBlock() | 367 |
| 7.66.2.2 FlashDone() | 368 |
| 7.66.2.3 FlashErase() | 368 |
| 7.66.2.4 Flash GetUserProgBaseAddress() | 368 |
| 7.66.2.5 FlashInit() | 369 |
| 7.66.2.6 FlashInitBlock() | 369 |
| 7.66.2.7 FlashSwitchBlock() | 369 |
| 7.66.2.8 FlashVerifyChecksum() | 370 |
| 7.66.2.9 FlashWrite() | 370 |
| 7.66.2.10 FlashWriteBlock() | 370 |
| 7.66.2.11 FlashWriteChecksum() | 371 |
| 7.66.3 Variable Documentation | 371 |
| 7.66.3.1 blockInfo | 371 |
| 7.66.3.2 bootBlockInfo | 372 |
| 7.66.3.3 flashLayout | 372 |
| 7.67 flash.c File Reference | 372 |

| | |
|---|-----|
| 7.67.1 Detailed Description | 374 |
| 7.67.2 Function Documentation | 374 |
| 7.67.2.1 FlashAddToBlock() | 374 |
| 7.67.2.2 FlashDone() | 375 |
| 7.67.2.3 FlashEmptyCheckSector() | 375 |
| 7.67.2.4 FlashErase() | 375 |
| 7.67.2.5 FlashEraseSectors() | 376 |
| 7.67.2.6 FlashGetSector() | 376 |
| 7.67.2.7 Flash GetUserProgBaseAddress() | 377 |
| 7.67.2.8 FlashInit() | 377 |
| 7.67.2.9 FlashInitBlock() | 377 |
| 7.67.2.10 FlashSwitchBlock() | 378 |
| 7.67.2.11 FlashVerifyChecksum() | 378 |
| 7.67.2.12 FlashWrite() | 378 |
| 7.67.2.13 FlashWriteBlock() | 379 |
| 7.67.2.14 FlashWriteChecksum() | 379 |
| 7.67.3 Variable Documentation | 379 |
| 7.67.3.1 blockInfo | 380 |
| 7.67.3.2 bootBlockInfo | 380 |
| 7.67.3.3 flashLayout | 380 |
| 7.68 flash.c File Reference | 381 |
| 7.68.1 Detailed Description | 383 |
| 7.68.2 Function Documentation | 383 |
| 7.68.2.1 FlashAddToBlock() | 383 |
| 7.68.2.2 FlashDone() | 383 |
| 7.68.2.3 FlashErase() | 384 |
| 7.68.2.4 FlashEraseSectors() | 384 |
| 7.68.2.5 FlashGetBank() | 384 |
| 7.68.2.6 FlashGetPage() | 385 |
| 7.68.2.7 FlashGetSectorIdx() | 385 |
| 7.68.2.8 Flash GetUserProgBaseAddress() | 386 |
| 7.68.2.9 FlashInit() | 386 |
| 7.68.2.10 FlashInitBlock() | 386 |
| 7.68.2.11 FlashSwitchBlock() | 387 |
| 7.68.2.12 FlashVerifyBankMode() | 387 |
| 7.68.2.13 FlashVerifyChecksum() | 387 |
| 7.68.2.14 FlashWrite() | 387 |
| 7.68.2.15 FlashWriteBlock() | 388 |
| 7.68.2.16 FlashWriteChecksum() | 388 |
| 7.68.3 Variable Documentation | 388 |
| 7.68.3.1 blockInfo | 389 |
| 7.68.3.2 bootBlockInfo | 389 |

| | |
|---|-----|
| 7.68.3.3 flashLayout | 389 |
| 7.69 flash.c File Reference | 390 |
| 7.69.1 Detailed Description | 391 |
| 7.69.2 Function Documentation | 392 |
| 7.69.2.1 FlashAddToBlock() | 392 |
| 7.69.2.2 FlashDone() | 392 |
| 7.69.2.3 FlashErase() | 392 |
| 7.69.2.4 FlashGetBank() | 393 |
| 7.69.2.5 FlashGetPage() | 393 |
| 7.69.2.6 Flash GetUserProgBaseAddress() | 394 |
| 7.69.2.7 FlashInit() | 394 |
| 7.69.2.8 FlashInitBlock() | 394 |
| 7.69.2.9 FlashSwitchBlock() | 395 |
| 7.69.2.10 FlashVerifyChecksum() | 395 |
| 7.69.2.11 FlashWrite() | 395 |
| 7.69.2.12 FlashWriteBlock() | 396 |
| 7.69.2.13 FlashWriteChecksum() | 396 |
| 7.69.3 Variable Documentation | 396 |
| 7.69.3.1 blockInfo | 397 |
| 7.69.3.2 bootBlockInfo | 397 |
| 7.69.3.3 flashLayout | 397 |
| 7.70 flash.c File Reference | 398 |
| 7.70.1 Detailed Description | 400 |
| 7.70.2 Function Documentation | 400 |
| 7.70.2.1 FlashAddToBlock() | 400 |
| 7.70.2.2 FlashDone() | 400 |
| 7.70.2.3 FlashErase() | 401 |
| 7.70.2.4 FlashEraseSectors() | 401 |
| 7.70.2.5 FlashGetSector() | 401 |
| 7.70.2.6 FlashGetSectorBaseAddr() | 402 |
| 7.70.2.7 FlashGetSectorSize() | 402 |
| 7.70.2.8 Flash GetUserProgBaseAddress() | 403 |
| 7.70.2.9 FlashInit() | 403 |
| 7.70.2.10 FlashInitBlock() | 403 |
| 7.70.2.11 FlashSwitchBlock() | 404 |
| 7.70.2.12 FlashVerifyChecksum() | 404 |
| 7.70.2.13 FlashWrite() | 404 |
| 7.70.2.14 FlashWriteBlock() | 405 |
| 7.70.2.15 FlashWriteChecksum() | 405 |
| 7.70.3 Variable Documentation | 405 |
| 7.70.3.1 blockInfo | 406 |
| 7.70.3.2 bootBlockInfo | 406 |

| | |
|--|-----|
| 7.70.3.3 flashLayout | 406 |
| 7.71 flash.c File Reference | 407 |
| 7.71.1 Detailed Description | 409 |
| 7.71.2 Function Documentation | 409 |
| 7.71.2.1 FlashAddToBlock() | 409 |
| 7.71.2.2 FlashDone() | 409 |
| 7.71.2.3 FlashEmptyCheckSector() | 410 |
| 7.71.2.4 FlashErase() | 411 |
| 7.71.2.5 FlashEraseSectors() | 411 |
| 7.71.2.6 FlashGetSector() | 412 |
| 7.71.2.7 FlashGetSectorBaseAddr() | 412 |
| 7.71.2.8 Flash GetUserProgBaseAddress() | 412 |
| 7.71.2.9 FlashInit() | 413 |
| 7.71.2.10 FlashInitBlock() | 413 |
| 7.71.2.11 FlashSwitchBlock() | 413 |
| 7.71.2.12 FlashTranslateToNonCachedAddress() | 414 |
| 7.71.2.13 FlashVerifyChecksum() | 414 |
| 7.71.2.14 FlashWrite() | 415 |
| 7.71.2.15 FlashWriteBlock() | 415 |
| 7.71.2.16 FlashWriteChecksum() | 415 |
| 7.71.3 Variable Documentation | 416 |
| 7.71.3.1 blockInfo | 416 |
| 7.71.3.2 bootBlockInfo | 416 |
| 7.71.3.3 flashLayout | 417 |
| 7.72 flash.c File Reference | 417 |
| 7.72.1 Detailed Description | 419 |
| 7.72.2 Function Documentation | 419 |
| 7.72.2.1 FlashAddToBlock() | 419 |
| 7.72.2.2 FlashDone() | 420 |
| 7.72.2.3 FlashEmptyCheckSector() | 420 |
| 7.72.2.4 FlashErase() | 421 |
| 7.72.2.5 FlashEraseSectors() | 421 |
| 7.72.2.6 FlashGetSector() | 421 |
| 7.72.2.7 Flash GetUserProgBaseAddress() | 422 |
| 7.72.2.8 FlashInit() | 422 |
| 7.72.2.9 FlashInitBlock() | 422 |
| 7.72.2.10 FlashIsSingleBankMode() | 423 |
| 7.72.2.11 FlashSwitchBlock() | 423 |
| 7.72.2.12 FlashVerifyChecksum() | 424 |
| 7.72.2.13 FlashWrite() | 424 |
| 7.72.2.14 FlashWriteBlock() | 424 |
| 7.72.2.15 FlashWriteChecksum() | 425 |

| | |
|---|-----|
| 7.72.3 Variable Documentation | 425 |
| 7.72.3.1 blockInfo | 425 |
| 7.72.3.2 bootBlockInfo | 425 |
| 7.72.3.3 flashLayout | 426 |
| 7.73 flash.c File Reference | 426 |
| 7.73.1 Detailed Description | 428 |
| 7.73.2 Function Documentation | 428 |
| 7.73.2.1 FlashAddToBlock() | 428 |
| 7.73.2.2 FlashDone() | 429 |
| 7.73.2.3 FlashEmptyCheckSector() | 429 |
| 7.73.2.4 FlashErase() | 429 |
| 7.73.2.5 FlashEraseSectors() | 430 |
| 7.73.2.6 FlashGetSectorIdx() | 430 |
| 7.73.2.7 Flash GetUserProgBaseAddress() | 430 |
| 7.73.2.8 FlashInit() | 431 |
| 7.73.2.9 FlashInitBlock() | 431 |
| 7.73.2.10 FlashSwitchBlock() | 431 |
| 7.73.2.11 FlashVerifyChecksum() | 432 |
| 7.73.2.12 FlashWrite() | 432 |
| 7.73.2.13 FlashWriteBlock() | 433 |
| 7.73.2.14 FlashWriteChecksum() | 433 |
| 7.73.3 Variable Documentation | 433 |
| 7.73.3.1 blockInfo | 433 |
| 7.73.3.2 bootBlockInfo | 434 |
| 7.73.3.3 flashLayout | 434 |
| 7.74 flash.c File Reference | 434 |
| 7.74.1 Detailed Description | 437 |
| 7.74.2 Function Documentation | 437 |
| 7.74.2.1 FlashAddToBlock() | 437 |
| 7.74.2.2 FlashDone() | 438 |
| 7.74.2.3 FlashErase() | 438 |
| 7.74.2.4 FlashExecuteCommand() | 439 |
| 7.74.2.5 FlashGetLinearAddrByte() | 439 |
| 7.74.2.6 FlashGetPhysAddr() | 439 |
| 7.74.2.7 FlashGetPhysPage() | 440 |
| 7.74.2.8 Flash GetUserProgBaseAddress() | 440 |
| 7.74.2.9 FlashInit() | 440 |
| 7.74.2.10 FlashInitBlock() | 441 |
| 7.74.2.11 FlashOperate() | 441 |
| 7.74.2.12 FlashSwitchBlock() | 441 |
| 7.74.2.13 FlashVerifyChecksum() | 442 |
| 7.74.2.14 FlashWrite() | 442 |

| | |
|--|-----|
| 7.74.2.15 FlashWriteBlock() | 443 |
| 7.74.2.16 FlashWriteChecksum() | 443 |
| 7.74.3 Variable Documentation | 443 |
| 7.74.3.1 blockInfo | 443 |
| 7.74.3.2 bootBlockInfo | 444 |
| 7.74.3.3 flashExecCmd | 444 |
| 7.74.3.4 flashLayout | 445 |
| 7.75 flash.c File Reference | 445 |
| 7.75.1 Detailed Description | 447 |
| 7.75.2 Function Documentation | 447 |
| 7.75.2.1 FlashAddToBlock() | 447 |
| 7.75.2.2 FlashDone() | 448 |
| 7.75.2.3 FlashErase() | 448 |
| 7.75.2.4 FlashEraseLogicalSector() | 449 |
| 7.75.2.5 FlashEraseSectors() | 449 |
| 7.75.2.6 FlashGetSectorIdx() | 449 |
| 7.75.2.7 Flash GetUserProgBaseAddress() | 450 |
| 7.75.2.8 FlashInit() | 450 |
| 7.75.2.9 FlashInitBlock() | 450 |
| 7.75.2.10 FlashSwitchBlock() | 451 |
| 7.75.2.11 FlashTranslateToNonCachedAddress() | 451 |
| 7.75.2.12 FlashVerifyChecksum() | 452 |
| 7.75.2.13 FlashWrite() | 452 |
| 7.75.2.14 FlashWriteBlock() | 452 |
| 7.75.2.15 FlashWriteChecksum() | 453 |
| 7.75.2.16 FlashWritePage() | 453 |
| 7.75.3 Variable Documentation | 454 |
| 7.75.3.1 blockInfo | 454 |
| 7.75.3.2 bootBlockInfo | 454 |
| 7.75.3.3 flashLayout | 454 |
| 7.76 flash.c File Reference | 455 |
| 7.76.1 Detailed Description | 457 |
| 7.76.2 Function Documentation | 457 |
| 7.76.2.1 FlashAddToBlock() | 457 |
| 7.76.2.2 FlashDone() | 457 |
| 7.76.2.3 FlashErase() | 458 |
| 7.76.2.4 FlashEraseLogicalSectors() | 458 |
| 7.76.2.5 FlashEraseSectors() | 458 |
| 7.76.2.6 FlashGetSectorIdx() | 460 |
| 7.76.2.7 Flash GetUserProgBaseAddress() | 460 |
| 7.76.2.8 FlashInit() | 461 |
| 7.76.2.9 FlashInitBlock() | 461 |

| | |
|---|-----|
| 7.76.2.10 FlashSwitchBlock() | 461 |
| 7.76.2.11 FlashVerifyChecksum() | 462 |
| 7.76.2.12 FlashWrite() | 462 |
| 7.76.2.13 FlashWriteBlock() | 462 |
| 7.76.2.14 FlashWriteChecksum() | 463 |
| 7.76.2.15 FlashWritePage() | 463 |
| 7.76.3 Variable Documentation | 464 |
| 7.76.3.1 blockInfo | 464 |
| 7.76.3.2 bootBlockInfo | 464 |
| 7.76.3.3 flashLayout | 464 |
| 7.77 flash.h File Reference | 465 |
| 7.77.1 Detailed Description | 465 |
| 7.77.2 Function Documentation | 465 |
| 7.77.2.1 FlashDone() | 466 |
| 7.77.2.2 FlashErase() | 466 |
| 7.77.2.3 Flash GetUserProgBaseAddress() | 466 |
| 7.77.2.4 FlashInit() | 467 |
| 7.77.2.5 FlashVerifyChecksum() | 467 |
| 7.77.2.6 FlashWrite() | 468 |
| 7.77.2.7 FlashWriteChecksum() | 468 |
| 7.78 flash.h File Reference | 468 |
| 7.78.1 Detailed Description | 469 |
| 7.78.2 Function Documentation | 469 |
| 7.78.2.1 FlashDone() | 469 |
| 7.78.2.2 FlashErase() | 469 |
| 7.78.2.3 Flash GetUserProgBaseAddress() | 470 |
| 7.78.2.4 FlashInit() | 471 |
| 7.78.2.5 FlashVerifyChecksum() | 471 |
| 7.78.2.6 FlashWrite() | 471 |
| 7.78.2.7 FlashWriteChecksum() | 472 |
| 7.79 flash.h File Reference | 472 |
| 7.79.1 Detailed Description | 473 |
| 7.79.2 Function Documentation | 473 |
| 7.79.2.1 FlashDone() | 473 |
| 7.79.2.2 FlashErase() | 473 |
| 7.79.2.3 Flash GetUserProgBaseAddress() | 474 |
| 7.79.2.4 FlashInit() | 475 |
| 7.79.2.5 FlashVerifyChecksum() | 475 |
| 7.79.2.6 FlashWrite() | 475 |
| 7.79.2.7 FlashWriteChecksum() | 476 |
| 7.80 flash.h File Reference | 476 |
| 7.80.1 Detailed Description | 477 |

| | |
|---|-----|
| 7.80.2 Function Documentation | 477 |
| 7.80.2.1 FlashDone() | 477 |
| 7.80.2.2 FlashErase() | 477 |
| 7.80.2.3 Flash GetUserProgBaseAddress() | 478 |
| 7.80.2.4 FlashInit() | 479 |
| 7.80.2.5 FlashVerifyChecksum() | 479 |
| 7.80.2.6 FlashWrite() | 479 |
| 7.80.2.7 FlashWriteChecksum() | 480 |
| 7.81 flash.h File Reference | 480 |
| 7.81.1 Detailed Description | 481 |
| 7.81.2 Function Documentation | 481 |
| 7.81.2.1 FlashDone() | 481 |
| 7.81.2.2 FlashErase() | 481 |
| 7.81.2.3 Flash GetUserProgBaseAddress() | 482 |
| 7.81.2.4 FlashInit() | 483 |
| 7.81.2.5 FlashVerifyChecksum() | 483 |
| 7.81.2.6 FlashWrite() | 483 |
| 7.81.2.7 FlashWriteChecksum() | 484 |
| 7.82 flash.h File Reference | 484 |
| 7.82.1 Detailed Description | 485 |
| 7.82.2 Function Documentation | 485 |
| 7.82.2.1 FlashDone() | 485 |
| 7.82.2.2 FlashErase() | 485 |
| 7.82.2.3 Flash GetUserProgBaseAddress() | 486 |
| 7.82.2.4 FlashInit() | 487 |
| 7.82.2.5 FlashVerifyChecksum() | 487 |
| 7.82.2.6 FlashWrite() | 487 |
| 7.82.2.7 FlashWriteChecksum() | 488 |
| 7.83 flash.h File Reference | 488 |
| 7.83.1 Detailed Description | 489 |
| 7.83.2 Function Documentation | 489 |
| 7.83.2.1 FlashDone() | 489 |
| 7.83.2.2 FlashErase() | 489 |
| 7.83.2.3 Flash GetUserProgBaseAddress() | 490 |
| 7.83.2.4 FlashInit() | 491 |
| 7.83.2.5 FlashVerifyChecksum() | 491 |
| 7.83.2.6 FlashWrite() | 491 |
| 7.83.2.7 FlashWriteChecksum() | 492 |
| 7.84 flash.h File Reference | 492 |
| 7.84.1 Detailed Description | 493 |
| 7.84.2 Function Documentation | 493 |
| 7.84.2.1 FlashDone() | 493 |

| | |
|---|-----|
| 7.84.2.2 FlashErase() | 493 |
| 7.84.2.3 Flash GetUserProgBaseAddress() | 494 |
| 7.84.2.4 FlashInit() | 495 |
| 7.84.2.5 FlashVerifyChecksum() | 495 |
| 7.84.2.6 FlashWrite() | 495 |
| 7.84.2.7 FlashWriteChecksum() | 496 |
| 7.85 flash.h File Reference | 496 |
| 7.85.1 Detailed Description | 497 |
| 7.85.2 Function Documentation | 497 |
| 7.85.2.1 FlashDone() | 497 |
| 7.85.2.2 FlashErase() | 497 |
| 7.85.2.3 Flash GetUserProgBaseAddress() | 498 |
| 7.85.2.4 FlashInit() | 499 |
| 7.85.2.5 FlashVerifyChecksum() | 499 |
| 7.85.2.6 FlashWrite() | 499 |
| 7.85.2.7 FlashWriteChecksum() | 500 |
| 7.86 flash.h File Reference | 500 |
| 7.86.1 Detailed Description | 501 |
| 7.86.2 Function Documentation | 501 |
| 7.86.2.1 FlashDone() | 501 |
| 7.86.2.2 FlashErase() | 501 |
| 7.86.2.3 Flash GetUserProgBaseAddress() | 502 |
| 7.86.2.4 FlashInit() | 503 |
| 7.86.2.5 FlashVerifyChecksum() | 503 |
| 7.86.2.6 FlashWrite() | 503 |
| 7.86.2.7 FlashWriteChecksum() | 504 |
| 7.87 flash.h File Reference | 504 |
| 7.87.1 Detailed Description | 505 |
| 7.87.2 Function Documentation | 505 |
| 7.87.2.1 FlashDone() | 505 |
| 7.87.2.2 FlashErase() | 505 |
| 7.87.2.3 Flash GetUserProgBaseAddress() | 506 |
| 7.87.2.4 FlashInit() | 507 |
| 7.87.2.5 FlashVerifyChecksum() | 507 |
| 7.87.2.6 FlashWrite() | 507 |
| 7.87.2.7 FlashWriteChecksum() | 508 |
| 7.88 flash.h File Reference | 508 |
| 7.88.1 Detailed Description | 509 |
| 7.88.2 Function Documentation | 509 |
| 7.88.2.1 FlashDone() | 509 |
| 7.88.2.2 FlashErase() | 509 |
| 7.88.2.3 Flash GetUserProgBaseAddress() | 510 |

| | |
|---|-----|
| 7.88.2.4 FlashInit() | 511 |
| 7.88.2.5 FlashVerifyChecksum() | 511 |
| 7.88.2.6 FlashWrite() | 511 |
| 7.88.2.7 FlashWriteChecksum() | 512 |
| 7.89 flash.h File Reference | 512 |
| 7.89.1 Detailed Description | 513 |
| 7.89.2 Function Documentation | 513 |
| 7.89.2.1 FlashDone() | 513 |
| 7.89.2.2 FlashErase() | 513 |
| 7.89.2.3 Flash GetUserProgBaseAddress() | 514 |
| 7.89.2.4 FlashInit() | 515 |
| 7.89.2.5 FlashVerifyChecksum() | 515 |
| 7.89.2.6 FlashWrite() | 515 |
| 7.89.2.7 FlashWriteChecksum() | 516 |
| 7.90 flash.h File Reference | 516 |
| 7.90.1 Detailed Description | 517 |
| 7.90.2 Function Documentation | 517 |
| 7.90.2.1 FlashDone() | 517 |
| 7.90.2.2 FlashErase() | 517 |
| 7.90.2.3 Flash GetUserProgBaseAddress() | 518 |
| 7.90.2.4 FlashInit() | 519 |
| 7.90.2.5 FlashVerifyChecksum() | 519 |
| 7.90.2.6 FlashWrite() | 519 |
| 7.90.2.7 FlashWriteChecksum() | 520 |
| 7.91 flash.h File Reference | 520 |
| 7.91.1 Detailed Description | 521 |
| 7.91.2 Function Documentation | 521 |
| 7.91.2.1 FlashDone() | 521 |
| 7.91.2.2 FlashErase() | 521 |
| 7.91.2.3 Flash GetUserProgBaseAddress() | 522 |
| 7.91.2.4 FlashInit() | 523 |
| 7.91.2.5 FlashVerifyChecksum() | 523 |
| 7.91.2.6 FlashWrite() | 523 |
| 7.91.2.7 FlashWriteChecksum() | 524 |
| 7.92 flash.h File Reference | 524 |
| 7.92.1 Detailed Description | 525 |
| 7.92.2 Function Documentation | 525 |
| 7.92.2.1 FlashDone() | 525 |
| 7.92.2.2 FlashErase() | 525 |
| 7.92.2.3 Flash GetUserProgBaseAddress() | 526 |
| 7.92.2.4 FlashInit() | 527 |
| 7.92.2.5 FlashVerifyChecksum() | 527 |

| | |
|---|-----|
| 7.92.2.6 FlashWrite() | 527 |
| 7.92.2.7 FlashWriteChecksum() | 528 |
| 7.93 flash.h File Reference | 528 |
| 7.93.1 Detailed Description | 529 |
| 7.93.2 Function Documentation | 529 |
| 7.93.2.1 FlashDone() | 529 |
| 7.93.2.2 FlashErase() | 529 |
| 7.93.2.3 Flash GetUserProgBaseAddress() | 530 |
| 7.93.2.4 FlashInit() | 531 |
| 7.93.2.5 FlashVerifyChecksum() | 531 |
| 7.93.2.6 FlashWrite() | 531 |
| 7.93.2.7 FlashWriteChecksum() | 532 |
| 7.94 flash.h File Reference | 532 |
| 7.94.1 Detailed Description | 533 |
| 7.94.2 Function Documentation | 533 |
| 7.94.2.1 FlashDone() | 533 |
| 7.94.2.2 FlashErase() | 533 |
| 7.94.2.3 Flash GetUserProgBaseAddress() | 534 |
| 7.94.2.4 FlashInit() | 535 |
| 7.94.2.5 FlashVerifyChecksum() | 535 |
| 7.94.2.6 FlashWrite() | 535 |
| 7.94.2.7 FlashWriteChecksum() | 536 |
| 7.95 flash.h File Reference | 536 |
| 7.95.1 Detailed Description | 537 |
| 7.95.2 Function Documentation | 537 |
| 7.95.2.1 FlashDone() | 537 |
| 7.95.2.2 FlashErase() | 537 |
| 7.95.2.3 Flash GetUserProgBaseAddress() | 538 |
| 7.95.2.4 FlashInit() | 539 |
| 7.95.2.5 FlashVerifyChecksum() | 539 |
| 7.95.2.6 FlashWrite() | 539 |
| 7.95.2.7 FlashWriteChecksum() | 540 |
| 7.96 flash.h File Reference | 540 |
| 7.96.1 Detailed Description | 541 |
| 7.96.2 Function Documentation | 541 |
| 7.96.2.1 FlashDone() | 541 |
| 7.96.2.2 FlashErase() | 541 |
| 7.96.2.3 Flash GetUserProgBaseAddress() | 542 |
| 7.96.2.4 FlashInit() | 543 |
| 7.96.2.5 FlashVerifyChecksum() | 543 |
| 7.96.2.6 FlashWrite() | 543 |
| 7.96.2.7 FlashWriteChecksum() | 544 |

| | |
|--|-----|
| 7.97 flash.h File Reference | 544 |
| 7.97.1 Detailed Description | 545 |
| 7.97.2 Function Documentation | 545 |
| 7.97.2.1 FlashDone() | 545 |
| 7.97.2.2 FlashErase() | 545 |
| 7.97.2.3 Flash GetUserProgBaseAddress() | 546 |
| 7.97.2.4 FlashInit() | 547 |
| 7.97.2.5 FlashVerifyChecksum() | 547 |
| 7.97.2.6 FlashWrite() | 547 |
| 7.97.2.7 FlashWriteChecksum() | 548 |
| 7.98 flash.h File Reference | 548 |
| 7.98.1 Detailed Description | 549 |
| 7.98.2 Function Documentation | 549 |
| 7.98.2.1 FlashDone() | 549 |
| 7.98.2.2 FlashErase() | 549 |
| 7.98.2.3 Flash GetUserProgBaseAddress() | 550 |
| 7.98.2.4 FlashInit() | 551 |
| 7.98.2.5 FlashVerifyChecksum() | 551 |
| 7.98.2.6 FlashWrite() | 551 |
| 7.98.2.7 FlashWriteChecksum() | 552 |
| 7.99 flash.h File Reference | 552 |
| 7.99.1 Detailed Description | 553 |
| 7.99.2 Function Documentation | 553 |
| 7.99.2.1 FlashDone() | 553 |
| 7.99.2.2 FlashErase() | 553 |
| 7.99.2.3 Flash GetUserProgBaseAddress() | 554 |
| 7.99.2.4 FlashInit() | 555 |
| 7.99.2.5 FlashVerifyChecksum() | 555 |
| 7.99.2.6 FlashWrite() | 555 |
| 7.99.2.7 FlashWriteChecksum() | 556 |
| 7.100 flash.h File Reference | 556 |
| 7.100.1 Detailed Description | 557 |
| 7.100.2 Function Documentation | 557 |
| 7.100.2.1 FlashDone() | 557 |
| 7.100.2.2 FlashErase() | 557 |
| 7.100.2.3 Flash GetUserProgBaseAddress() | 558 |
| 7.100.2.4 FlashInit() | 559 |
| 7.100.2.5 FlashVerifyChecksum() | 559 |
| 7.100.2.6 FlashWrite() | 559 |
| 7.100.2.7 FlashWriteChecksum() | 560 |
| 7.101 flash.h File Reference | 560 |
| 7.101.1 Detailed Description | 561 |

| | |
|--|-----|
| 7.101.2 Function Documentation | 561 |
| 7.101.2.1 FlashDone() | 561 |
| 7.101.2.2 FlashErase() | 561 |
| 7.101.2.3 Flash GetUserProgBaseAddress() | 562 |
| 7.101.2.4 FlashInit() | 563 |
| 7.101.2.5 FlashVerifyChecksum() | 563 |
| 7.101.2.6 FlashWrite() | 563 |
| 7.101.2.7 FlashWriteChecksum() | 564 |
| 7.102 flash.h File Reference | 564 |
| 7.102.1 Detailed Description | 565 |
| 7.102.2 Function Documentation | 565 |
| 7.102.2.1 FlashDone() | 565 |
| 7.102.2.2 FlashErase() | 565 |
| 7.102.2.3 Flash GetUserProgBaseAddress() | 566 |
| 7.102.2.4 FlashInit() | 567 |
| 7.102.2.5 FlashVerifyChecksum() | 567 |
| 7.102.2.6 FlashWrite() | 567 |
| 7.102.2.7 FlashWriteChecksum() | 568 |
| 7.103 flash.h File Reference | 568 |
| 7.103.1 Detailed Description | 569 |
| 7.103.2 Function Documentation | 569 |
| 7.103.2.1 FlashDone() | 569 |
| 7.103.2.2 FlashErase() | 569 |
| 7.103.2.3 Flash GetUserProgBaseAddress() | 570 |
| 7.103.2.4 FlashInit() | 571 |
| 7.103.2.5 FlashVerifyChecksum() | 571 |
| 7.103.2.6 FlashWrite() | 571 |
| 7.103.2.7 FlashWriteChecksum() | 572 |
| 7.104 cpu_comp.c File Reference | 572 |
| 7.104.1 Detailed Description | 573 |
| 7.104.2 Function Documentation | 573 |
| 7.104.2.1 CpuIrqDisable() | 573 |
| 7.104.2.2 CpuIrqEnable() | 573 |
| 7.105 cpu_comp.c File Reference | 573 |
| 7.105.1 Detailed Description | 574 |
| 7.105.2 Function Documentation | 574 |
| 7.105.2.1 CpuIrqDisable() | 574 |
| 7.105.2.2 CpuIrqEnable() | 574 |
| 7.106 cpu_comp.c File Reference | 575 |
| 7.106.1 Detailed Description | 575 |
| 7.106.2 Function Documentation | 575 |
| 7.106.2.1 CpuIrqDisable() | 575 |

| | |
|---------------------------------|-----|
| 7.106.2.2 CpuIrqEnable() | 576 |
| 7.107 cpu_comp.c File Reference | 576 |
| 7.107.1 Detailed Description | 576 |
| 7.107.2 Function Documentation | 576 |
| 7.107.2.1 CpuIrqDisable() | 577 |
| 7.107.2.2 CpuIrqEnable() | 577 |
| 7.108 cpu_comp.c File Reference | 577 |
| 7.108.1 Detailed Description | 578 |
| 7.108.2 Function Documentation | 578 |
| 7.108.2.1 CpuIrqDisable() | 578 |
| 7.108.2.2 CpuIrqEnable() | 578 |
| 7.109 cpu_comp.c File Reference | 578 |
| 7.109.1 Detailed Description | 579 |
| 7.109.2 Function Documentation | 579 |
| 7.109.2.1 CpuIrqDisable() | 579 |
| 7.109.2.2 CpuIrqEnable() | 579 |
| 7.110 cpu_comp.c File Reference | 580 |
| 7.110.1 Detailed Description | 580 |
| 7.110.2 Function Documentation | 580 |
| 7.110.2.1 CpuIrqDisable() | 580 |
| 7.110.2.2 CpuIrqEnable() | 581 |
| 7.111 cpu_comp.c File Reference | 581 |
| 7.111.1 Detailed Description | 581 |
| 7.111.2 Function Documentation | 581 |
| 7.111.2.1 CpuIrqDisable() | 582 |
| 7.111.2.2 CpuIrqEnable() | 582 |
| 7.112 cpu_comp.c File Reference | 582 |
| 7.112.1 Detailed Description | 583 |
| 7.112.2 Function Documentation | 583 |
| 7.112.2.1 CpuIrqDisable() | 583 |
| 7.112.2.2 CpuIrqEnable() | 583 |
| 7.113 cpu_comp.c File Reference | 583 |
| 7.113.1 Detailed Description | 584 |
| 7.113.2 Function Documentation | 584 |
| 7.113.2.1 CpuIrqDisable() | 584 |
| 7.113.2.2 CpuIrqEnable() | 584 |
| 7.114 cpu_comp.c File Reference | 585 |
| 7.114.1 Detailed Description | 585 |
| 7.114.2 Function Documentation | 585 |
| 7.114.2.1 CpuIrqDisable() | 585 |
| 7.114.2.2 CpuIrqEnable() | 586 |
| 7.115 cpu_comp.c File Reference | 586 |

| | |
|---|-----|
| 7.115.1 Detailed Description | 586 |
| 7.115.2 Function Documentation | 586 |
| 7.115.2.1 CpuIrqDisable() | 587 |
| 7.115.2.2 CpuIrqEnable() | 587 |
| 7.116 cpu_comp.c File Reference | 587 |
| 7.116.1 Detailed Description | 588 |
| 7.116.2 Function Documentation | 588 |
| 7.116.2.1 CpuIrqDisable() | 588 |
| 7.116.2.2 CpuIrqEnable() | 588 |
| 7.117 cpu_comp.c File Reference | 588 |
| 7.117.1 Detailed Description | 589 |
| 7.117.2 Function Documentation | 589 |
| 7.117.2.1 CpuIrqDisable() | 589 |
| 7.117.2.2 CpuIrqEnable() | 589 |
| 7.118 cpu_comp.c File Reference | 590 |
| 7.118.1 Detailed Description | 590 |
| 7.118.2 Function Documentation | 590 |
| 7.118.2.1 CpuIrqDisable() | 590 |
| 7.118.2.2 CpuIrqEnable() | 591 |
| 7.119 cpu_comp.c File Reference | 591 |
| 7.119.1 Detailed Description | 591 |
| 7.119.2 Function Documentation | 591 |
| 7.119.2.1 CpuIrqDisable() | 592 |
| 7.119.2.2 CpuIrqEnable() | 592 |
| 7.120 cpu_comp.c File Reference | 592 |
| 7.120.1 Detailed Description | 593 |
| 7.120.2 Function Documentation | 593 |
| 7.120.2.1 CpuIrqDisable() | 593 |
| 7.120.2.2 CpuIrqEnable() | 593 |
| 7.121 cpu_comp.c File Reference | 593 |
| 7.121.1 Detailed Description | 594 |
| 7.121.2 Function Documentation | 594 |
| 7.121.2.1 CpuIrqDisable() | 594 |
| 7.121.2.2 CpuIrqEnable() | 594 |
| 7.122 cpu_comp.c File Reference | 595 |
| 7.122.1 Detailed Description | 595 |
| 7.122.2 Function Documentation | 595 |
| 7.122.2.1 CpuIrqDisable() | 595 |
| 7.122.2.2 CpuIrqEnable() | 596 |
| 7.123 cpu_comp.c File Reference | 596 |
| 7.123.1 Detailed Description | 596 |
| 7.123.2 Function Documentation | 596 |

| | |
|---------------------------------|-----|
| 7.123.2.1 CpuIrqDisable() | 597 |
| 7.123.2.2 CpuIrqEnable() | 597 |
| 7.124 cpu_comp.c File Reference | 597 |
| 7.124.1 Detailed Description | 598 |
| 7.124.2 Function Documentation | 598 |
| 7.124.2.1 CpuIrqDisable() | 598 |
| 7.124.2.2 CpuIrqEnable() | 598 |
| 7.125 cpu_comp.c File Reference | 598 |
| 7.125.1 Detailed Description | 599 |
| 7.125.2 Function Documentation | 599 |
| 7.125.2.1 CpuIrqDisable() | 599 |
| 7.125.2.2 CpuIrqEnable() | 599 |
| 7.126 cpu_comp.c File Reference | 600 |
| 7.126.1 Detailed Description | 600 |
| 7.126.2 Function Documentation | 600 |
| 7.126.2.1 CpuIrqDisable() | 600 |
| 7.126.2.2 CpuIrqEnable() | 601 |
| 7.127 cpu_comp.c File Reference | 601 |
| 7.127.1 Detailed Description | 601 |
| 7.127.2 Function Documentation | 601 |
| 7.127.2.1 CpuIrqDisable() | 602 |
| 7.127.2.2 CpuIrqEnable() | 602 |
| 7.128 cpu_comp.c File Reference | 602 |
| 7.128.1 Detailed Description | 603 |
| 7.128.2 Function Documentation | 603 |
| 7.128.2.1 CpuIrqDisable() | 603 |
| 7.128.2.2 CpuIrqEnable() | 603 |
| 7.129 cpu_comp.c File Reference | 603 |
| 7.129.1 Detailed Description | 604 |
| 7.129.2 Function Documentation | 604 |
| 7.129.2.1 CpuIrqDisable() | 604 |
| 7.129.2.2 CpuIrqEnable() | 604 |
| 7.130 cpu_comp.c File Reference | 605 |
| 7.130.1 Detailed Description | 605 |
| 7.130.2 Function Documentation | 605 |
| 7.130.2.1 CpuIrqDisable() | 605 |
| 7.130.2.2 CpuIrqEnable() | 606 |
| 7.131 cpu_comp.c File Reference | 606 |
| 7.131.1 Detailed Description | 606 |
| 7.131.2 Function Documentation | 606 |
| 7.131.2.1 CpuIrqDisable() | 607 |
| 7.131.2.2 CpuIrqEnable() | 607 |

| | |
|---|-----|
| 7.132 cpu_comp.c File Reference | 607 |
| 7.132.1 Detailed Description | 608 |
| 7.132.2 Function Documentation | 608 |
| 7.132.2.1 CpuIrqDisable() | 608 |
| 7.132.2.2 CpuIrqEnable() | 608 |
| 7.133 cpu_comp.c File Reference | 608 |
| 7.133.1 Detailed Description | 609 |
| 7.133.2 Function Documentation | 609 |
| 7.133.2.1 CpuIrqDisable() | 609 |
| 7.133.2.2 CpuIrqEnable() | 609 |
| 7.134 cpu_comp.c File Reference | 610 |
| 7.134.1 Detailed Description | 610 |
| 7.134.2 Function Documentation | 610 |
| 7.134.2.1 CpuIrqDisable() | 610 |
| 7.134.2.2 CpuIrqEnable() | 611 |
| 7.135 cpu_comp.c File Reference | 611 |
| 7.135.1 Detailed Description | 611 |
| 7.135.2 Function Documentation | 611 |
| 7.135.2.1 CpuIrqDisable() | 612 |
| 7.135.2.2 CpuIrqEnable() | 612 |
| 7.136 cpu_comp.c File Reference | 612 |
| 7.136.1 Detailed Description | 613 |
| 7.136.2 Function Documentation | 613 |
| 7.136.2.1 CpuIrqDisable() | 613 |
| 7.136.2.2 CpuIrqEnable() | 613 |
| 7.137 cpu_comp.c File Reference | 613 |
| 7.137.1 Detailed Description | 614 |
| 7.137.2 Function Documentation | 614 |
| 7.137.2.1 CpuIrqDisable() | 614 |
| 7.137.2.2 CpuIrqEnable() | 614 |
| 7.138 cpu_comp.c File Reference | 615 |
| 7.138.1 Detailed Description | 615 |
| 7.138.2 Function Documentation | 615 |
| 7.138.2.1 CpuIrqDisable() | 615 |
| 7.138.2.2 CpuIrqEnable() | 616 |
| 7.139 cpu_comp.c File Reference | 616 |
| 7.139.1 Detailed Description | 616 |
| 7.139.2 Function Documentation | 616 |
| 7.139.2.1 CpuIrqDisable() | 617 |
| 7.139.2.2 CpuIrqEnable() | 617 |
| 7.140 cpu_comp.c File Reference | 617 |
| 7.140.1 Detailed Description | 618 |

| | |
|---|-----|
| 7.140.2 Function Documentation | 618 |
| 7.140.2.1 CpuIrqDisable() | 618 |
| 7.140.2.2 CpuIrqEnable() | 618 |
| 7.141 cpu_comp.c File Reference | 618 |
| 7.141.1 Detailed Description | 619 |
| 7.141.2 Function Documentation | 619 |
| 7.141.2.1 CpuIrqDisable() | 619 |
| 7.141.2.2 CpuIrqEnable() | 619 |
| 7.142 cpu_comp.c File Reference | 620 |
| 7.142.1 Detailed Description | 620 |
| 7.142.2 Function Documentation | 620 |
| 7.142.2.1 CpuIrqDisable() | 620 |
| 7.142.2.2 CpuIrqEnable() | 621 |
| 7.143 cpu_comp.c File Reference | 621 |
| 7.143.1 Detailed Description | 621 |
| 7.143.2 Function Documentation | 621 |
| 7.143.2.1 CpuIrqDisable() | 622 |
| 7.143.2.2 CpuIrqEnable() | 622 |
| 7.144 cpu_comp.c File Reference | 622 |
| 7.144.1 Detailed Description | 623 |
| 7.144.2 Function Documentation | 623 |
| 7.144.2.1 CpuIrqDisable() | 623 |
| 7.144.2.2 CpuIrqEnable() | 623 |
| 7.145 cpu_comp.c File Reference | 623 |
| 7.145.1 Detailed Description | 624 |
| 7.145.2 Function Documentation | 624 |
| 7.145.2.1 CpuIrqDisable() | 624 |
| 7.145.2.2 CpuIrqEnable() | 624 |
| 7.146 cpu_comp.c File Reference | 625 |
| 7.146.1 Detailed Description | 625 |
| 7.146.2 Function Documentation | 625 |
| 7.146.2.1 CpuIrqDisable() | 625 |
| 7.146.2.2 CpuIrqEnable() | 626 |
| 7.147 cpu_comp.c File Reference | 626 |
| 7.147.1 Detailed Description | 626 |
| 7.147.2 Function Documentation | 626 |
| 7.147.2.1 CpuIrqDisable() | 627 |
| 7.147.2.2 CpuIrqEnable() | 627 |
| 7.148 cpu_comp.c File Reference | 627 |
| 7.148.1 Detailed Description | 628 |
| 7.148.2 Function Documentation | 628 |
| 7.148.2.1 CpuIrqDisable() | 628 |

| | |
|---------------------------------|-----|
| 7.148.2.2 CpuIrqEnable() | 628 |
| 7.149 cpu_comp.c File Reference | 628 |
| 7.149.1 Detailed Description | 629 |
| 7.149.2 Function Documentation | 629 |
| 7.149.2.1 CpuIrqDisable() | 629 |
| 7.149.2.2 CpuIrqEnable() | 629 |
| 7.150 cpu_comp.c File Reference | 630 |
| 7.150.1 Detailed Description | 630 |
| 7.150.2 Function Documentation | 630 |
| 7.150.2.1 CpuIrqDisable() | 630 |
| 7.150.2.2 CpuIrqEnable() | 631 |
| 7.151 cpu_comp.c File Reference | 631 |
| 7.151.1 Detailed Description | 631 |
| 7.151.2 Function Documentation | 631 |
| 7.151.2.1 CpuIrqDisable() | 632 |
| 7.151.2.2 CpuIrqEnable() | 632 |
| 7.152 cpu_comp.c File Reference | 632 |
| 7.152.1 Detailed Description | 633 |
| 7.152.2 Function Documentation | 633 |
| 7.152.2.1 CpuIrqDisable() | 633 |
| 7.152.2.2 CpuIrqEnable() | 633 |
| 7.153 cpu_comp.c File Reference | 633 |
| 7.153.1 Detailed Description | 634 |
| 7.153.2 Function Documentation | 634 |
| 7.153.2.1 CpuIrqDisable() | 634 |
| 7.153.2.2 CpuIrqEnable() | 634 |
| 7.154 cpu_comp.c File Reference | 635 |
| 7.154.1 Detailed Description | 635 |
| 7.154.2 Function Documentation | 635 |
| 7.154.2.1 CpuIrqDisable() | 635 |
| 7.154.2.2 CpuIrqEnable() | 636 |
| 7.155 cpu_comp.c File Reference | 636 |
| 7.155.1 Detailed Description | 636 |
| 7.155.2 Function Documentation | 636 |
| 7.155.2.1 CpuIrqDisable() | 637 |
| 7.155.2.2 CpuIrqEnable() | 637 |
| 7.156 cpu_comp.c File Reference | 637 |
| 7.156.1 Detailed Description | 638 |
| 7.156.2 Function Documentation | 638 |
| 7.156.2.1 CpuIrqDisable() | 638 |
| 7.156.2.2 CpuIrqEnable() | 638 |
| 7.157 cpu_comp.c File Reference | 638 |

| | |
|---|-----|
| 7.157.1 Detailed Description | 639 |
| 7.157.2 Function Documentation | 639 |
| 7.157.2.1 CpuIrqDisable() | 639 |
| 7.157.2.2 CpuIrqEnable() | 639 |
| 7.158 cpu_comp.c File Reference | 640 |
| 7.158.1 Detailed Description | 640 |
| 7.158.2 Function Documentation | 640 |
| 7.158.2.1 CpuIrqDisable() | 640 |
| 7.158.2.2 CpuIrqEnable() | 641 |
| 7.159 cpu_comp.c File Reference | 641 |
| 7.159.1 Detailed Description | 641 |
| 7.159.2 Function Documentation | 641 |
| 7.159.2.1 CpuIrqDisable() | 642 |
| 7.159.2.2 CpuIrqEnable() | 642 |
| 7.160 cpu_comp.c File Reference | 642 |
| 7.160.1 Detailed Description | 643 |
| 7.160.2 Function Documentation | 643 |
| 7.160.2.1 CpuIrqDisable() | 643 |
| 7.160.2.2 CpuIrqEnable() | 643 |
| 7.161 cpu_comp.c File Reference | 643 |
| 7.161.1 Detailed Description | 644 |
| 7.161.2 Function Documentation | 644 |
| 7.161.2.1 CpuIrqDisable() | 644 |
| 7.161.2.2 CpuIrqEnable() | 644 |
| 7.162 cpu_comp.c File Reference | 645 |
| 7.162.1 Detailed Description | 645 |
| 7.162.2 Function Documentation | 645 |
| 7.162.2.1 CpuIrqDisable() | 645 |
| 7.162.2.2 CpuIrqEnable() | 646 |
| 7.163 cpu_comp.c File Reference | 646 |
| 7.163.1 Detailed Description | 646 |
| 7.163.2 Function Documentation | 646 |
| 7.163.2.1 CpuIrqDisable() | 647 |
| 7.163.2.2 CpuIrqEnable() | 647 |
| 7.164 cpu_comp.c File Reference | 647 |
| 7.164.1 Detailed Description | 648 |
| 7.164.2 Function Documentation | 648 |
| 7.164.2.1 CpuIrqDisable() | 648 |
| 7.164.2.2 CpuIrqEnable() | 648 |
| 7.165 cpu_comp.c File Reference | 648 |
| 7.165.1 Detailed Description | 649 |
| 7.165.2 Function Documentation | 649 |

| | |
|---------------------------------|-----|
| 7.165.2.1 CpuIrqDisable() | 649 |
| 7.165.2.2 CpuIrqEnable() | 649 |
| 7.166 cpu_comp.c File Reference | 650 |
| 7.166.1 Detailed Description | 650 |
| 7.166.2 Function Documentation | 650 |
| 7.166.2.1 CpuIrqDisable() | 650 |
| 7.166.2.2 CpuIrqEnable() | 651 |
| 7.167 cpu_comp.c File Reference | 651 |
| 7.167.1 Detailed Description | 651 |
| 7.167.2 Function Documentation | 651 |
| 7.167.2.1 CpuIrqDisable() | 652 |
| 7.167.2.2 CpuIrqEnable() | 652 |
| 7.168 cpu_comp.c File Reference | 652 |
| 7.168.1 Detailed Description | 653 |
| 7.168.2 Function Documentation | 653 |
| 7.168.2.1 CpuIrqDisable() | 653 |
| 7.168.2.2 CpuIrqEnable() | 653 |
| 7.169 cpu_comp.c File Reference | 653 |
| 7.169.1 Detailed Description | 654 |
| 7.169.2 Function Documentation | 654 |
| 7.169.2.1 CpuIrqDisable() | 654 |
| 7.169.2.2 CpuIrqEnable() | 654 |
| 7.170 cpu_comp.c File Reference | 655 |
| 7.170.1 Detailed Description | 655 |
| 7.170.2 Function Documentation | 655 |
| 7.170.2.1 CpuIrqDisable() | 655 |
| 7.170.2.2 CpuIrqEnable() | 656 |
| 7.171 mbrtu.c File Reference | 656 |
| 7.171.1 Detailed Description | 656 |
| 7.172 mbrtu.c File Reference | 656 |
| 7.172.1 Detailed Description | 657 |
| 7.173 mbrtu.c File Reference | 657 |
| 7.173.1 Detailed Description | 657 |
| 7.174 mbrtu.c File Reference | 657 |
| 7.174.1 Detailed Description | 658 |
| 7.175 mbrtu.c File Reference | 658 |
| 7.175.1 Detailed Description | 658 |
| 7.176 mbrtu.c File Reference | 658 |
| 7.176.1 Detailed Description | 659 |
| 7.177 mbrtu.c File Reference | 659 |
| 7.177.1 Detailed Description | 659 |
| 7.178 mbrtu.c File Reference | 659 |

| | |
|--|-----|
| 7.178.1 Detailed Description | 660 |
| 7.179 mbrtu.c File Reference | 660 |
| 7.179.1 Detailed Description | 660 |
| 7.180 mbrtu.c File Reference | 660 |
| 7.180.1 Detailed Description | 661 |
| 7.181 mbrtu.c File Reference | 661 |
| 7.181.1 Detailed Description | 661 |
| 7.182 mbrtu.c File Reference | 661 |
| 7.182.1 Detailed Description | 662 |
| 7.183 mbrtu.c File Reference | 662 |
| 7.183.1 Detailed Description | 662 |
| 7.184 mbrtu.c File Reference | 662 |
| 7.184.1 Detailed Description | 663 |
| 7.185 mbrtu.c File Reference | 663 |
| 7.185.1 Detailed Description | 663 |
| 7.186 mbrtu.c File Reference | 663 |
| 7.186.1 Detailed Description | 664 |
| 7.187 mbrtu.c File Reference | 664 |
| 7.187.1 Detailed Description | 664 |
| 7.188 mbrtu.c File Reference | 664 |
| 7.188.1 Detailed Description | 665 |
| 7.189 mbrtu.c File Reference | 665 |
| 7.189.1 Detailed Description | 665 |
| 7.190 mbrtu.c File Reference | 665 |
| 7.190.1 Detailed Description | 666 |
| 7.191 mbrtu.c File Reference | 666 |
| 7.191.1 Detailed Description | 666 |
| 7.192 mbrtu.c File Reference | 666 |
| 7.192.1 Detailed Description | 667 |
| 7.193 mbrtu.c File Reference | 667 |
| 7.193.1 Detailed Description | 667 |
| 7.194 mbrtu.c File Reference | 667 |
| 7.194.1 Detailed Description | 668 |
| 7.195 mbrtu.c File Reference | 668 |
| 7.195.1 Detailed Description | 668 |
| 7.196 mbrtu.c File Reference | 668 |
| 7.196.1 Detailed Description | 669 |
| 7.197 mbrtu.c File Reference | 669 |
| 7.197.1 Detailed Description | 669 |
| 7.198 nvm.c File Reference | 669 |
| 7.198.1 Detailed Description | 670 |
| 7.198.2 Function Documentation | 670 |

| | |
|--|-----|
| 7.198.2.1 NvmDone() | 670 |
| 7.198.2.2 NvmErase() | 670 |
| 7.198.2.3 Nvm GetUserProgBaseAddress() | 671 |
| 7.198.2.4 NvmInit() | 671 |
| 7.198.2.5 NvmVerifyChecksum() | 671 |
| 7.198.2.6 NvmWrite() | 672 |
| 7.199 nvm.c File Reference | 673 |
| 7.199.1 Detailed Description | 674 |
| 7.199.2 Function Documentation | 674 |
| 7.199.2.1 NvmDone() | 674 |
| 7.199.2.2 NvmErase() | 674 |
| 7.199.2.3 Nvm GetUserProgBaseAddress() | 674 |
| 7.199.2.4 NvmInit() | 675 |
| 7.199.2.5 NvmVerifyChecksum() | 675 |
| 7.199.2.6 NvmWrite() | 675 |
| 7.200 nvm.c File Reference | 676 |
| 7.200.1 Detailed Description | 676 |
| 7.200.2 Function Documentation | 676 |
| 7.200.2.1 NvmDone() | 677 |
| 7.200.2.2 NvmErase() | 677 |
| 7.200.2.3 Nvm GetUserProgBaseAddress() | 677 |
| 7.200.2.4 NvmInit() | 678 |
| 7.200.2.5 NvmVerifyChecksum() | 678 |
| 7.200.2.6 NvmWrite() | 678 |
| 7.201 nvm.c File Reference | 678 |
| 7.201.1 Detailed Description | 679 |
| 7.201.2 Function Documentation | 679 |
| 7.201.2.1 NvmDone() | 680 |
| 7.201.2.2 NvmErase() | 680 |
| 7.201.2.3 Nvm GetUserProgBaseAddress() | 680 |
| 7.201.2.4 NvmInit() | 681 |
| 7.201.2.5 NvmVerifyChecksum() | 681 |
| 7.201.2.6 NvmWrite() | 681 |
| 7.202 nvm.c File Reference | 681 |
| 7.202.1 Detailed Description | 682 |
| 7.202.2 Function Documentation | 682 |
| 7.202.2.1 NvmDone() | 683 |
| 7.202.2.2 NvmErase() | 683 |
| 7.202.2.3 Nvm GetUserProgBaseAddress() | 683 |
| 7.202.2.4 NvmInit() | 684 |
| 7.202.2.5 NvmVerifyChecksum() | 684 |
| 7.202.2.6 NvmWrite() | 684 |

| | |
|--|-----|
| 7.203 nvm.c File Reference | 684 |
| 7.203.1 Detailed Description | 685 |
| 7.203.2 Function Documentation | 685 |
| 7.203.2.1 NvmDone() | 686 |
| 7.203.2.2 NvmErase() | 686 |
| 7.203.2.3 Nvm GetUserProgBaseAddress() | 686 |
| 7.203.2.4 NvmInit() | 687 |
| 7.203.2.5 NvmVerifyChecksum() | 687 |
| 7.203.2.6 NvmWrite() | 687 |
| 7.204 nvm.c File Reference | 687 |
| 7.204.1 Detailed Description | 688 |
| 7.204.2 Function Documentation | 688 |
| 7.204.2.1 NvmDone() | 689 |
| 7.204.2.2 NvmErase() | 689 |
| 7.204.2.3 Nvm GetUserProgBaseAddress() | 689 |
| 7.204.2.4 NvmInit() | 690 |
| 7.204.2.5 NvmVerifyChecksum() | 690 |
| 7.204.2.6 NvmWrite() | 690 |
| 7.205 nvm.c File Reference | 690 |
| 7.205.1 Detailed Description | 691 |
| 7.205.2 Function Documentation | 691 |
| 7.205.2.1 NvmDone() | 692 |
| 7.205.2.2 NvmErase() | 692 |
| 7.205.2.3 Nvm GetUserProgBaseAddress() | 692 |
| 7.205.2.4 NvmInit() | 693 |
| 7.205.2.5 NvmVerifyChecksum() | 693 |
| 7.205.2.6 NvmWrite() | 693 |
| 7.206 nvm.c File Reference | 693 |
| 7.206.1 Detailed Description | 694 |
| 7.206.2 Function Documentation | 694 |
| 7.206.2.1 NvmDone() | 695 |
| 7.206.2.2 NvmErase() | 695 |
| 7.206.2.3 Nvm GetUserProgBaseAddress() | 695 |
| 7.206.2.4 NvmInit() | 696 |
| 7.206.2.5 NvmVerifyChecksum() | 696 |
| 7.206.2.6 NvmWrite() | 696 |
| 7.207 nvm.c File Reference | 696 |
| 7.207.1 Detailed Description | 697 |
| 7.207.2 Function Documentation | 697 |
| 7.207.2.1 NvmDone() | 698 |
| 7.207.2.2 NvmErase() | 698 |
| 7.207.2.3 Nvm GetUserProgBaseAddress() | 698 |

| | |
|--|-----|
| 7.207.2.4 NvmInit() | 699 |
| 7.207.2.5 NvmVerifyChecksum() | 699 |
| 7.207.2.6 NvmWrite() | 699 |
| 7.208 nvm.c File Reference | 699 |
| 7.208.1 Detailed Description | 700 |
| 7.208.2 Function Documentation | 700 |
| 7.208.2.1 NvmDone() | 701 |
| 7.208.2.2 NvmErase() | 701 |
| 7.208.2.3 Nvm GetUserProgBaseAddress() | 701 |
| 7.208.2.4 NvmInit() | 702 |
| 7.208.2.5 NvmVerifyChecksum() | 702 |
| 7.208.2.6 NvmWrite() | 702 |
| 7.209 nvm.c File Reference | 702 |
| 7.209.1 Detailed Description | 703 |
| 7.209.2 Function Documentation | 703 |
| 7.209.2.1 NvmDone() | 704 |
| 7.209.2.2 NvmErase() | 704 |
| 7.209.2.3 Nvm GetUserProgBaseAddress() | 704 |
| 7.209.2.4 NvmInit() | 705 |
| 7.209.2.5 NvmVerifyChecksum() | 705 |
| 7.209.2.6 NvmWrite() | 705 |
| 7.210 nvm.c File Reference | 705 |
| 7.210.1 Detailed Description | 706 |
| 7.210.2 Function Documentation | 706 |
| 7.210.2.1 NvmDone() | 707 |
| 7.210.2.2 NvmErase() | 707 |
| 7.210.2.3 Nvm GetUserProgBaseAddress() | 707 |
| 7.210.2.4 NvmInit() | 708 |
| 7.210.2.5 NvmVerifyChecksum() | 708 |
| 7.210.2.6 NvmWrite() | 708 |
| 7.211 nvm.c File Reference | 708 |
| 7.211.1 Detailed Description | 709 |
| 7.211.2 Function Documentation | 709 |
| 7.211.2.1 NvmDone() | 710 |
| 7.211.2.2 NvmErase() | 710 |
| 7.211.2.3 Nvm GetUserProgBaseAddress() | 710 |
| 7.211.2.4 NvmInit() | 711 |
| 7.211.2.5 NvmVerifyChecksum() | 711 |
| 7.211.2.6 NvmWrite() | 711 |
| 7.212 nvm.c File Reference | 711 |
| 7.212.1 Detailed Description | 712 |
| 7.212.2 Function Documentation | 712 |

| | |
|--|-----|
| 7.212.2.1 NvmDone() | 713 |
| 7.212.2.2 NvmErase() | 713 |
| 7.212.2.3 Nvm GetUserProgBaseAddress() | 713 |
| 7.212.2.4 NvmInit() | 714 |
| 7.212.2.5 NvmVerifyChecksum() | 714 |
| 7.212.2.6 NvmWrite() | 714 |
| 7.213 nvm.c File Reference | 714 |
| 7.213.1 Detailed Description | 715 |
| 7.213.2 Function Documentation | 715 |
| 7.213.2.1 NvmDone() | 716 |
| 7.213.2.2 NvmErase() | 716 |
| 7.213.2.3 Nvm GetUserProgBaseAddress() | 716 |
| 7.213.2.4 NvmInit() | 717 |
| 7.213.2.5 NvmVerifyChecksum() | 717 |
| 7.213.2.6 NvmWrite() | 717 |
| 7.214 nvm.c File Reference | 717 |
| 7.214.1 Detailed Description | 718 |
| 7.214.2 Function Documentation | 718 |
| 7.214.2.1 NvmDone() | 719 |
| 7.214.2.2 NvmErase() | 719 |
| 7.214.2.3 Nvm GetUserProgBaseAddress() | 719 |
| 7.214.2.4 NvmInit() | 720 |
| 7.214.2.5 NvmVerifyChecksum() | 720 |
| 7.214.2.6 NvmWrite() | 720 |
| 7.215 nvm.c File Reference | 720 |
| 7.215.1 Detailed Description | 721 |
| 7.215.2 Function Documentation | 721 |
| 7.215.2.1 NvmDone() | 722 |
| 7.215.2.2 NvmErase() | 722 |
| 7.215.2.3 Nvm GetUserProgBaseAddress() | 722 |
| 7.215.2.4 NvmInit() | 723 |
| 7.215.2.5 NvmVerifyChecksum() | 723 |
| 7.215.2.6 NvmWrite() | 723 |
| 7.216 nvm.c File Reference | 723 |
| 7.216.1 Detailed Description | 724 |
| 7.216.2 Function Documentation | 724 |
| 7.216.2.1 NvmDone() | 725 |
| 7.216.2.2 NvmErase() | 725 |
| 7.216.2.3 Nvm GetUserProgBaseAddress() | 725 |
| 7.216.2.4 NvmInit() | 726 |
| 7.216.2.5 NvmVerifyChecksum() | 726 |
| 7.216.2.6 NvmWrite() | 726 |

| | |
|--|-----|
| 7.217 nvm.c File Reference | 726 |
| 7.217.1 Detailed Description | 727 |
| 7.217.2 Function Documentation | 727 |
| 7.217.2.1 NvmDone() | 728 |
| 7.217.2.2 NvmErase() | 728 |
| 7.217.2.3 Nvm GetUserProgBaseAddress() | 728 |
| 7.217.2.4 NvmInit() | 729 |
| 7.217.2.5 NvmVerifyChecksum() | 729 |
| 7.217.2.6 NvmWrite() | 729 |
| 7.218 nvm.c File Reference | 729 |
| 7.218.1 Detailed Description | 730 |
| 7.218.2 Function Documentation | 730 |
| 7.218.2.1 NvmDone() | 731 |
| 7.218.2.2 NvmErase() | 731 |
| 7.218.2.3 Nvm GetUserProgBaseAddress() | 731 |
| 7.218.2.4 NvmInit() | 732 |
| 7.218.2.5 NvmVerifyChecksum() | 732 |
| 7.218.2.6 NvmWrite() | 732 |
| 7.219 nvm.c File Reference | 732 |
| 7.219.1 Detailed Description | 733 |
| 7.219.2 Function Documentation | 733 |
| 7.219.2.1 NvmDone() | 734 |
| 7.219.2.2 NvmErase() | 734 |
| 7.219.2.3 Nvm GetUserProgBaseAddress() | 734 |
| 7.219.2.4 NvmInit() | 735 |
| 7.219.2.5 NvmVerifyChecksum() | 735 |
| 7.219.2.6 NvmWrite() | 735 |
| 7.220 nvm.c File Reference | 735 |
| 7.220.1 Detailed Description | 736 |
| 7.220.2 Function Documentation | 736 |
| 7.220.2.1 NvmDone() | 737 |
| 7.220.2.2 NvmErase() | 737 |
| 7.220.2.3 Nvm GetUserProgBaseAddress() | 737 |
| 7.220.2.4 NvmInit() | 738 |
| 7.220.2.5 NvmVerifyChecksum() | 738 |
| 7.220.2.6 NvmWrite() | 738 |
| 7.221 nvm.c File Reference | 738 |
| 7.221.1 Detailed Description | 739 |
| 7.221.2 Function Documentation | 739 |
| 7.221.2.1 NvmDone() | 740 |
| 7.221.2.2 NvmErase() | 740 |
| 7.221.2.3 Nvm GetUserProgBaseAddress() | 740 |

| | |
|--|-----|
| 7.221.2.4 NvmInit() | 741 |
| 7.221.2.5 NvmVerifyChecksum() | 741 |
| 7.221.2.6 NvmWrite() | 741 |
| 7.222 nvm.c File Reference | 741 |
| 7.222.1 Detailed Description | 742 |
| 7.222.2 Function Documentation | 742 |
| 7.222.2.1 NvmDone() | 743 |
| 7.222.2.2 NvmErase() | 743 |
| 7.222.2.3 Nvm GetUserProgBaseAddress() | 743 |
| 7.222.2.4 NvmInit() | 744 |
| 7.222.2.5 NvmVerifyChecksum() | 744 |
| 7.222.2.6 NvmWrite() | 744 |
| 7.223 nvm.c File Reference | 744 |
| 7.223.1 Detailed Description | 745 |
| 7.223.2 Function Documentation | 745 |
| 7.223.2.1 NvmDone() | 746 |
| 7.223.2.2 NvmErase() | 746 |
| 7.223.2.3 Nvm GetUserProgBaseAddress() | 746 |
| 7.223.2.4 NvmInit() | 747 |
| 7.223.2.5 NvmVerifyChecksum() | 747 |
| 7.223.2.6 NvmWrite() | 747 |
| 7.224 nvm.c File Reference | 747 |
| 7.224.1 Detailed Description | 748 |
| 7.224.2 Function Documentation | 748 |
| 7.224.2.1 NvmDone() | 749 |
| 7.224.2.2 NvmErase() | 749 |
| 7.224.2.3 Nvm GetUserProgBaseAddress() | 749 |
| 7.224.2.4 NvmInit() | 750 |
| 7.224.2.5 NvmVerifyChecksum() | 750 |
| 7.224.2.6 NvmWrite() | 750 |
| 7.225 rs232.c File Reference | 750 |
| 7.225.1 Detailed Description | 751 |
| 7.226 rs232.c File Reference | 751 |
| 7.226.1 Detailed Description | 751 |
| 7.227 rs232.c File Reference | 752 |
| 7.227.1 Detailed Description | 752 |
| 7.228 rs232.c File Reference | 752 |
| 7.228.1 Detailed Description | 752 |
| 7.229 rs232.c File Reference | 753 |
| 7.229.1 Detailed Description | 753 |
| 7.230 rs232.c File Reference | 753 |
| 7.230.1 Detailed Description | 753 |

| | |
|--|-----|
| 7.231 rs232.c File Reference | 754 |
| 7.231.1 Detailed Description | 754 |
| 7.232 rs232.c File Reference | 754 |
| 7.232.1 Detailed Description | 754 |
| 7.233 rs232.c File Reference | 755 |
| 7.233.1 Detailed Description | 755 |
| 7.234 rs232.c File Reference | 755 |
| 7.234.1 Detailed Description | 755 |
| 7.235 rs232.c File Reference | 756 |
| 7.235.1 Detailed Description | 756 |
| 7.236 rs232.c File Reference | 756 |
| 7.236.1 Detailed Description | 756 |
| 7.237 rs232.c File Reference | 757 |
| 7.237.1 Detailed Description | 757 |
| 7.238 rs232.c File Reference | 757 |
| 7.238.1 Detailed Description | 757 |
| 7.239 rs232.c File Reference | 758 |
| 7.239.1 Detailed Description | 758 |
| 7.240 rs232.c File Reference | 758 |
| 7.240.1 Detailed Description | 758 |
| 7.241 rs232.c File Reference | 759 |
| 7.241.1 Detailed Description | 759 |
| 7.242 rs232.c File Reference | 759 |
| 7.242.1 Detailed Description | 759 |
| 7.243 rs232.c File Reference | 760 |
| 7.243.1 Detailed Description | 760 |
| 7.244 rs232.c File Reference | 760 |
| 7.244.1 Detailed Description | 760 |
| 7.245 rs232.c File Reference | 761 |
| 7.245.1 Detailed Description | 761 |
| 7.246 rs232.c File Reference | 761 |
| 7.246.1 Detailed Description | 762 |
| 7.247 rs232.c File Reference | 762 |
| 7.247.1 Detailed Description | 762 |
| 7.248 rs232.c File Reference | 762 |
| 7.248.1 Detailed Description | 763 |
| 7.249 rs232.c File Reference | 763 |
| 7.249.1 Detailed Description | 763 |
| 7.250 rs232.c File Reference | 763 |
| 7.250.1 Detailed Description | 764 |
| 7.251 rs232.c File Reference | 764 |
| 7.251.1 Detailed Description | 764 |

| | |
|--|-----|
| 7.252 timer.c File Reference | 764 |
| 7.252.1 Detailed Description | 765 |
| 7.252.2 Function Documentation | 765 |
| 7.252.2.1 TimerGet() | 765 |
| 7.252.2.2 TimerInit() | 766 |
| 7.252.2.3 TimerReset() | 766 |
| 7.252.2.4 TimerUpdate() | 766 |
| 7.253 timer.c File Reference | 767 |
| 7.253.1 Detailed Description | 767 |
| 7.253.2 Function Documentation | 767 |
| 7.253.2.1 TimerGet() | 768 |
| 7.253.2.2 TimerInit() | 768 |
| 7.253.2.3 TimerReset() | 768 |
| 7.253.2.4 TimerUpdate() | 769 |
| 7.254 timer.c File Reference | 769 |
| 7.254.1 Detailed Description | 770 |
| 7.254.2 Function Documentation | 770 |
| 7.254.2.1 HAL_GetTick() | 770 |
| 7.254.2.2 TimerGet() | 770 |
| 7.254.2.3 TimerInit() | 771 |
| 7.254.2.4 TimerReset() | 771 |
| 7.254.2.5 TimerUpdate() | 771 |
| 7.255 timer.c File Reference | 772 |
| 7.255.1 Detailed Description | 773 |
| 7.255.2 Function Documentation | 773 |
| 7.255.2.1 HAL_GetTick() | 773 |
| 7.255.2.2 TimerGet() | 773 |
| 7.255.2.3 TimerInit() | 773 |
| 7.255.2.4 TimerReset() | 774 |
| 7.255.2.5 TimerUpdate() | 774 |
| 7.256 timer.c File Reference | 774 |
| 7.256.1 Detailed Description | 775 |
| 7.256.2 Function Documentation | 775 |
| 7.256.2.1 HAL_GetTick() | 775 |
| 7.256.2.2 TimerGet() | 776 |
| 7.256.2.3 TimerInit() | 776 |
| 7.256.2.4 TimerReset() | 776 |
| 7.256.2.5 TimerUpdate() | 777 |
| 7.257 timer.c File Reference | 777 |
| 7.257.1 Detailed Description | 778 |
| 7.257.2 Function Documentation | 778 |
| 7.257.2.1 HAL_GetTick() | 778 |

| | |
|--------------------------------|-----|
| 7.257.2.2 TimerGet() | 778 |
| 7.257.2.3 TimerInit() | 779 |
| 7.257.2.4 TimerReset() | 779 |
| 7.257.2.5 TimerUpdate() | 779 |
| 7.258 timer.c File Reference | 780 |
| 7.258.1 Detailed Description | 780 |
| 7.258.2 Function Documentation | 780 |
| 7.258.2.1 TimerGet() | 781 |
| 7.258.2.2 TimerInit() | 781 |
| 7.258.2.3 TimerReset() | 781 |
| 7.258.2.4 TimerUpdate() | 782 |
| 7.259 timer.c File Reference | 782 |
| 7.259.1 Detailed Description | 783 |
| 7.259.2 Function Documentation | 783 |
| 7.259.2.1 HAL_GetTick() | 783 |
| 7.259.2.2 TimerGet() | 783 |
| 7.259.2.3 TimerInit() | 784 |
| 7.259.2.4 TimerReset() | 784 |
| 7.259.2.5 TimerUpdate() | 784 |
| 7.260 timer.c File Reference | 785 |
| 7.260.1 Detailed Description | 786 |
| 7.260.2 Function Documentation | 786 |
| 7.260.2.1 HAL_GetTick() | 786 |
| 7.260.2.2 TimerGet() | 786 |
| 7.260.2.3 TimerInit() | 786 |
| 7.260.2.4 TimerReset() | 787 |
| 7.260.2.5 TimerUpdate() | 787 |
| 7.261 timer.c File Reference | 787 |
| 7.261.1 Detailed Description | 788 |
| 7.261.2 Function Documentation | 788 |
| 7.261.2.1 HAL_GetTick() | 788 |
| 7.261.2.2 TimerGet() | 789 |
| 7.261.2.3 TimerInit() | 789 |
| 7.261.2.4 TimerReset() | 789 |
| 7.261.2.5 TimerUpdate() | 790 |
| 7.262 timer.c File Reference | 790 |
| 7.262.1 Detailed Description | 791 |
| 7.262.2 Function Documentation | 791 |
| 7.262.2.1 TimerGet() | 791 |
| 7.262.2.2 TimerInit() | 791 |
| 7.262.2.3 TimerReset() | 791 |
| 7.262.2.4 TimerUpdate() | 792 |

| | |
|--|-----|
| 7.263 timer.c File Reference | 792 |
| 7.263.1 Detailed Description | 793 |
| 7.263.2 Function Documentation | 793 |
| 7.263.2.1 TimerGet() | 793 |
| 7.263.2.2 TimerInit() | 793 |
| 7.263.2.3 TimerReset() | 793 |
| 7.263.2.4 TimerUpdate() | 794 |
| 7.264 timer.c File Reference | 794 |
| 7.264.1 Detailed Description | 795 |
| 7.264.2 Function Documentation | 795 |
| 7.264.2.1 HAL_GetTick() | 795 |
| 7.264.2.2 TimerGet() | 795 |
| 7.264.2.3 TimerInit() | 796 |
| 7.264.2.4 TimerReset() | 796 |
| 7.264.2.5 TimerUpdate() | 796 |
| 7.265 timer.c File Reference | 797 |
| 7.265.1 Detailed Description | 798 |
| 7.265.2 Function Documentation | 798 |
| 7.265.2.1 HAL_GetTick() | 798 |
| 7.265.2.2 TimerGet() | 798 |
| 7.265.2.3 TimerInit() | 798 |
| 7.265.2.4 TimerReset() | 799 |
| 7.265.2.5 TimerUpdate() | 799 |
| 7.266 timer.c File Reference | 799 |
| 7.266.1 Detailed Description | 800 |
| 7.266.2 Function Documentation | 800 |
| 7.266.2.1 HAL_GetTick() | 800 |
| 7.266.2.2 TimerGet() | 801 |
| 7.266.2.3 TimerInit() | 801 |
| 7.266.2.4 TimerReset() | 801 |
| 7.266.2.5 TimerUpdate() | 802 |
| 7.267 timer.c File Reference | 802 |
| 7.267.1 Detailed Description | 803 |
| 7.267.2 Function Documentation | 803 |
| 7.267.2.1 TimerGet() | 803 |
| 7.267.2.2 TimerInit() | 803 |
| 7.267.2.3 TimerReset() | 803 |
| 7.267.2.4 TimerUpdate() | 804 |
| 7.268 timer.c File Reference | 804 |
| 7.268.1 Detailed Description | 805 |
| 7.268.2 Function Documentation | 805 |
| 7.268.2.1 HAL_GetTick() | 805 |

| | |
|--------------------------------|-----|
| 7.268.2.2 TimerGet() | 805 |
| 7.268.2.3 TimerInit() | 806 |
| 7.268.2.4 TimerReset() | 806 |
| 7.268.2.5 TimerUpdate() | 806 |
| 7.269 timer.c File Reference | 807 |
| 7.269.1 Detailed Description | 808 |
| 7.269.2 Function Documentation | 808 |
| 7.269.2.1 HAL_GetTick() | 808 |
| 7.269.2.2 TimerGet() | 808 |
| 7.269.2.3 TimerInit() | 808 |
| 7.269.2.4 TimerReset() | 809 |
| 7.269.2.5 TimerUpdate() | 809 |
| 7.270 timer.c File Reference | 809 |
| 7.270.1 Detailed Description | 810 |
| 7.270.2 Function Documentation | 810 |
| 7.270.2.1 HAL_GetTick() | 810 |
| 7.270.2.2 TimerGet() | 811 |
| 7.270.2.3 TimerInit() | 811 |
| 7.270.2.4 TimerReset() | 811 |
| 7.270.2.5 TimerUpdate() | 812 |
| 7.271 timer.c File Reference | 812 |
| 7.271.1 Detailed Description | 813 |
| 7.271.2 Function Documentation | 813 |
| 7.271.2.1 HAL_GetTick() | 813 |
| 7.271.2.2 TimerGet() | 813 |
| 7.271.2.3 TimerInit() | 814 |
| 7.271.2.4 TimerReset() | 814 |
| 7.271.2.5 TimerUpdate() | 814 |
| 7.272 timer.c File Reference | 815 |
| 7.272.1 Detailed Description | 815 |
| 7.272.2 Function Documentation | 815 |
| 7.272.2.1 TimerGet() | 816 |
| 7.272.2.2 TimerInit() | 816 |
| 7.272.2.3 TimerReset() | 816 |
| 7.272.2.4 TimerUpdate() | 816 |
| 7.273 timer.c File Reference | 817 |
| 7.273.1 Detailed Description | 817 |
| 7.273.2 Function Documentation | 817 |
| 7.273.2.1 TimerGet() | 818 |
| 7.273.2.2 TimerInit() | 818 |
| 7.273.2.3 TimerReset() | 818 |
| 7.273.2.4 TimerUpdate() | 819 |

| | |
|--|-----|
| 7.274 timer.c File Reference | 819 |
| 7.274.1 Detailed Description | 820 |
| 7.274.2 Function Documentation | 820 |
| 7.274.2.1 HAL_GetTick() | 820 |
| 7.274.2.2 TimerGet() | 820 |
| 7.274.2.3 TimerInit() | 821 |
| 7.274.2.4 TimerReset() | 821 |
| 7.274.2.5 TimerUpdate() | 821 |
| 7.275 timer.c File Reference | 822 |
| 7.275.1 Detailed Description | 823 |
| 7.275.2 Function Documentation | 823 |
| 7.275.2.1 HAL_GetTick() | 823 |
| 7.275.2.2 TimerGet() | 823 |
| 7.275.2.3 TimerInit() | 823 |
| 7.275.2.4 TimerReset() | 824 |
| 7.275.2.5 TimerUpdate() | 824 |
| 7.276 timer.c File Reference | 824 |
| 7.276.1 Detailed Description | 825 |
| 7.276.2 Function Documentation | 825 |
| 7.276.2.1 TimerGet() | 826 |
| 7.276.2.2 TimerInit() | 826 |
| 7.276.2.3 TimerReset() | 826 |
| 7.276.2.4 TimerUpdate() | 826 |
| 7.277 timer.c File Reference | 827 |
| 7.277.1 Detailed Description | 827 |
| 7.277.2 Function Documentation | 827 |
| 7.277.2.1 TimerGet() | 828 |
| 7.277.2.2 TimerInit() | 828 |
| 7.277.2.3 TimerReset() | 828 |
| 7.277.2.4 TimerUpdate() | 828 |
| 7.278 timer.c File Reference | 829 |
| 7.278.1 Detailed Description | 829 |
| 7.278.2 Function Documentation | 829 |
| 7.278.2.1 TimerGet() | 830 |
| 7.278.2.2 TimerInit() | 830 |
| 7.278.2.3 TimerReset() | 830 |
| 7.278.2.4 TimerUpdate() | 830 |
| 7.279 types.h File Reference | 831 |
| 7.279.1 Detailed Description | 831 |
| 7.279.2 Typedef Documentation | 831 |
| 7.279.2.1 blt_addr | 831 |
| 7.279.2.2 blt_bool | 831 |

| | |
|---|-----|
| 7.279.2.3 blt_char | 832 |
| 7.279.2.4 blt_int16s | 832 |
| 7.279.2.5 blt_int16u | 832 |
| 7.279.2.6 blt_int32s | 832 |
| 7.279.2.7 blt_int32u | 832 |
| 7.279.2.8 blt_int8s | 832 |
| 7.279.2.9 blt_int8u | 832 |
| 7.280 types.h File Reference | 833 |
| 7.280.1 Detailed Description | 833 |
| 7.280.2 Typedef Documentation | 833 |
| 7.280.2.1 blt_addr | 833 |
| 7.280.2.2 blt_bool | 833 |
| 7.280.2.3 blt_char | 834 |
| 7.280.2.4 blt_int16s | 834 |
| 7.280.2.5 blt_int16u | 834 |
| 7.280.2.6 blt_int32s | 834 |
| 7.280.2.7 blt_int32u | 834 |
| 7.280.2.8 blt_int8s | 834 |
| 7.280.2.9 blt_int8u | 834 |
| 7.281 types.h File Reference | 835 |
| 7.281.1 Detailed Description | 835 |
| 7.281.2 Typedef Documentation | 835 |
| 7.281.2.1 blt_addr | 835 |
| 7.281.2.2 blt_bool | 836 |
| 7.281.2.3 blt_char | 836 |
| 7.281.2.4 blt_int16s | 836 |
| 7.281.2.5 blt_int16u | 836 |
| 7.281.2.6 blt_int32s | 836 |
| 7.281.2.7 blt_int32u | 836 |
| 7.281.2.8 blt_int64s | 836 |
| 7.281.2.9 blt_int64u | 837 |
| 7.281.2.10 blt_int8s | 837 |
| 7.281.2.11 blt_int8u | 837 |
| 7.282 types.h File Reference | 837 |
| 7.282.1 Detailed Description | 838 |
| 7.282.2 Typedef Documentation | 838 |
| 7.282.2.1 blt_addr | 838 |
| 7.282.2.2 blt_bool | 838 |
| 7.282.2.3 blt_char | 838 |
| 7.282.2.4 blt_int16s | 838 |
| 7.282.2.5 blt_int16u | 838 |
| 7.282.2.6 blt_int32s | 839 |

| | |
|---|-----|
| 7.282.2.7 blt_int32u | 839 |
| 7.282.2.8 blt_int8s | 839 |
| 7.282.2.9 blt_int8u | 839 |
| 7.283 types.h File Reference | 839 |
| 7.283.1 Detailed Description | 840 |
| 7.283.2 Typedef Documentation | 840 |
| 7.283.2.1 blt_addr | 840 |
| 7.283.2.2 blt_bool | 840 |
| 7.283.2.3 blt_char | 840 |
| 7.283.2.4 blt_int16s | 841 |
| 7.283.2.5 blt_int16u | 841 |
| 7.283.2.6 blt_int32s | 841 |
| 7.283.2.7 blt_int32u | 841 |
| 7.283.2.8 blt_int64s | 841 |
| 7.283.2.9 blt_int64u | 841 |
| 7.283.2.10 blt_int8s | 841 |
| 7.283.2.11 blt_int8u | 842 |
| 7.284 types.h File Reference | 842 |
| 7.284.1 Detailed Description | 842 |
| 7.284.2 Typedef Documentation | 842 |
| 7.284.2.1 blt_addr | 843 |
| 7.284.2.2 blt_bool | 843 |
| 7.284.2.3 blt_char | 843 |
| 7.284.2.4 blt_int16s | 843 |
| 7.284.2.5 blt_int16u | 843 |
| 7.284.2.6 blt_int32s | 843 |
| 7.284.2.7 blt_int32u | 843 |
| 7.284.2.8 blt_int8s | 844 |
| 7.284.2.9 blt_int8u | 844 |
| 7.285 types.h File Reference | 844 |
| 7.285.1 Detailed Description | 844 |
| 7.285.2 Typedef Documentation | 845 |
| 7.285.2.1 blt_addr | 845 |
| 7.285.2.2 blt_bool | 845 |
| 7.285.2.3 blt_char | 845 |
| 7.285.2.4 blt_int16s | 845 |
| 7.285.2.5 blt_int16u | 845 |
| 7.285.2.6 blt_int32s | 845 |
| 7.285.2.7 blt_int32u | 846 |
| 7.285.2.8 blt_int8s | 846 |
| 7.285.2.9 blt_int8u | 846 |
| 7.286 types.h File Reference | 846 |

| | |
|---|-----|
| 7.286.1 Detailed Description | 847 |
| 7.286.2 Typedef Documentation | 847 |
| 7.286.2.1 blt_addr | 847 |
| 7.286.2.2 blt_bool | 847 |
| 7.286.2.3 blt_char | 847 |
| 7.286.2.4 blt_int16s | 847 |
| 7.286.2.5 blt_int16u | 847 |
| 7.286.2.6 blt_int32s | 848 |
| 7.286.2.7 blt_int32u | 848 |
| 7.286.2.8 blt_int64s | 848 |
| 7.286.2.9 blt_int64u | 848 |
| 7.286.2.10 blt_int8s | 848 |
| 7.286.2.11 blt_int8u | 848 |
| 7.287 types.h File Reference | 848 |
| 7.287.1 Detailed Description | 849 |
| 7.287.2 Typedef Documentation | 849 |
| 7.287.2.1 blt_addr | 849 |
| 7.287.2.2 blt_bool | 849 |
| 7.287.2.3 blt_char | 850 |
| 7.287.2.4 blt_int16s | 850 |
| 7.287.2.5 blt_int16u | 850 |
| 7.287.2.6 blt_int32s | 850 |
| 7.287.2.7 blt_int32u | 850 |
| 7.287.2.8 blt_int64s | 850 |
| 7.287.2.9 blt_int64u | 850 |
| 7.287.2.10 blt_int8s | 851 |
| 7.287.2.11 blt_int8u | 851 |
| 7.288 types.h File Reference | 851 |
| 7.288.1 Detailed Description | 851 |
| 7.288.2 Typedef Documentation | 852 |
| 7.288.2.1 blt_addr | 852 |
| 7.288.2.2 blt_bool | 852 |
| 7.288.2.3 blt_char | 852 |
| 7.288.2.4 blt_int16s | 852 |
| 7.288.2.5 blt_int16u | 852 |
| 7.288.2.6 blt_int32s | 852 |
| 7.288.2.7 blt_int32u | 853 |
| 7.288.2.8 blt_int64s | 853 |
| 7.288.2.9 blt_int64u | 853 |
| 7.288.2.10 blt_int8s | 853 |
| 7.288.2.11 blt_int8u | 853 |
| 7.289 types.h File Reference | 853 |

| | |
|---|-----|
| 7.289.1 Detailed Description | 854 |
| 7.289.2 Typedef Documentation | 854 |
| 7.289.2.1 blt_addr | 854 |
| 7.289.2.2 blt_bool | 854 |
| 7.289.2.3 blt_char | 854 |
| 7.289.2.4 blt_int16s | 854 |
| 7.289.2.5 blt_int16u | 855 |
| 7.289.2.6 blt_int32s | 855 |
| 7.289.2.7 blt_int32u | 855 |
| 7.289.2.8 blt_int8s | 855 |
| 7.289.2.9 blt_int8u | 855 |
| 7.290 types.h File Reference | 855 |
| 7.290.1 Detailed Description | 856 |
| 7.290.2 Typedef Documentation | 856 |
| 7.290.2.1 blt_addr | 856 |
| 7.290.2.2 blt_bool | 856 |
| 7.290.2.3 blt_char | 856 |
| 7.290.2.4 blt_int16s | 856 |
| 7.290.2.5 blt_int16u | 857 |
| 7.290.2.6 blt_int32s | 857 |
| 7.290.2.7 blt_int32u | 857 |
| 7.290.2.8 blt_int8s | 857 |
| 7.290.2.9 blt_int8u | 857 |
| 7.291 types.h File Reference | 857 |
| 7.291.1 Detailed Description | 858 |
| 7.291.2 Typedef Documentation | 858 |
| 7.291.2.1 blt_addr | 858 |
| 7.291.2.2 blt_bool | 858 |
| 7.291.2.3 blt_char | 858 |
| 7.291.2.4 blt_int16s | 858 |
| 7.291.2.5 blt_int16u | 859 |
| 7.291.2.6 blt_int32s | 859 |
| 7.291.2.7 blt_int32u | 859 |
| 7.291.2.8 blt_int8s | 859 |
| 7.291.2.9 blt_int8u | 859 |
| 7.292 types.h File Reference | 859 |
| 7.292.1 Detailed Description | 860 |
| 7.292.2 Typedef Documentation | 860 |
| 7.292.2.1 blt_addr | 860 |
| 7.292.2.2 blt_bool | 860 |
| 7.292.2.3 blt_char | 860 |
| 7.292.2.4 blt_int16s | 860 |

| | |
|---|-----|
| 7.292.2.5 blt_int16u | 861 |
| 7.292.2.6 blt_int32s | 861 |
| 7.292.2.7 blt_int32u | 861 |
| 7.292.2.8 blt_int8s | 861 |
| 7.292.2.9 blt_int8u | 861 |
| 7.293 types.h File Reference | 861 |
| 7.293.1 Detailed Description | 862 |
| 7.293.2 Typedef Documentation | 862 |
| 7.293.2.1 blt_addr | 862 |
| 7.293.2.2 blt_bool | 862 |
| 7.293.2.3 blt_char | 862 |
| 7.293.2.4 blt_int16s | 862 |
| 7.293.2.5 blt_int16u | 863 |
| 7.293.2.6 blt_int32s | 863 |
| 7.293.2.7 blt_int32u | 863 |
| 7.293.2.8 blt_int8s | 863 |
| 7.293.2.9 blt_int8u | 863 |
| 7.294 types.h File Reference | 863 |
| 7.294.1 Detailed Description | 864 |
| 7.294.2 Typedef Documentation | 864 |
| 7.294.2.1 blt_addr | 864 |
| 7.294.2.2 blt_bool | 864 |
| 7.294.2.3 blt_char | 864 |
| 7.294.2.4 blt_int16s | 864 |
| 7.294.2.5 blt_int16u | 865 |
| 7.294.2.6 blt_int32s | 865 |
| 7.294.2.7 blt_int32u | 865 |
| 7.294.2.8 blt_int8s | 865 |
| 7.294.2.9 blt_int8u | 865 |
| 7.295 types.h File Reference | 865 |
| 7.295.1 Detailed Description | 866 |
| 7.295.2 Typedef Documentation | 866 |
| 7.295.2.1 blt_addr | 866 |
| 7.295.2.2 blt_bool | 866 |
| 7.295.2.3 blt_char | 866 |
| 7.295.2.4 blt_int16s | 866 |
| 7.295.2.5 blt_int16u | 867 |
| 7.295.2.6 blt_int32s | 867 |
| 7.295.2.7 blt_int32u | 867 |
| 7.295.2.8 blt_int8s | 867 |
| 7.295.2.9 blt_int8u | 867 |
| 7.296 types.h File Reference | 867 |

| | |
|---|-----|
| 7.296.1 Detailed Description | 868 |
| 7.296.2 Typedef Documentation | 868 |
| 7.296.2.1 blt_addr | 868 |
| 7.296.2.2 blt_bool | 868 |
| 7.296.2.3 blt_char | 868 |
| 7.296.2.4 blt_int16s | 868 |
| 7.296.2.5 blt_int16u | 869 |
| 7.296.2.6 blt_int32s | 869 |
| 7.296.2.7 blt_int32u | 869 |
| 7.296.2.8 blt_int8s | 869 |
| 7.296.2.9 blt_int8u | 869 |
| 7.297 types.h File Reference | 869 |
| 7.297.1 Detailed Description | 870 |
| 7.297.2 Typedef Documentation | 870 |
| 7.297.2.1 blt_addr | 870 |
| 7.297.2.2 blt_bool | 870 |
| 7.297.2.3 blt_char | 870 |
| 7.297.2.4 blt_int16s | 871 |
| 7.297.2.5 blt_int16u | 871 |
| 7.297.2.6 blt_int32s | 871 |
| 7.297.2.7 blt_int32u | 871 |
| 7.297.2.8 blt_int64s | 871 |
| 7.297.2.9 blt_int64u | 871 |
| 7.297.2.10 blt_int8s | 871 |
| 7.297.2.11 blt_int8u | 872 |
| 7.298 types.h File Reference | 872 |
| 7.298.1 Detailed Description | 872 |
| 7.298.2 Typedef Documentation | 872 |
| 7.298.2.1 blt_addr | 873 |
| 7.298.2.2 blt_bool | 873 |
| 7.298.2.3 blt_char | 873 |
| 7.298.2.4 blt_int16s | 873 |
| 7.298.2.5 blt_int16u | 873 |
| 7.298.2.6 blt_int32s | 873 |
| 7.298.2.7 blt_int32u | 873 |
| 7.298.2.8 blt_int8s | 874 |
| 7.298.2.9 blt_int8u | 874 |
| 7.299 types.h File Reference | 874 |
| 7.299.1 Detailed Description | 874 |
| 7.299.2 Typedef Documentation | 875 |
| 7.299.2.1 blt_addr | 875 |
| 7.299.2.2 blt_bool | 875 |

| | |
|---|-----|
| 7.299.2.3 blt_char | 875 |
| 7.299.2.4 blt_int16s | 875 |
| 7.299.2.5 blt_int16u | 875 |
| 7.299.2.6 blt_int32s | 875 |
| 7.299.2.7 blt_int32u | 876 |
| 7.299.2.8 blt_int8s | 876 |
| 7.299.2.9 blt_int8u | 876 |
| 7.300 types.h File Reference | 876 |
| 7.300.1 Detailed Description | 877 |
| 7.300.2 Typedef Documentation | 877 |
| 7.300.2.1 blt_addr | 877 |
| 7.300.2.2 blt_bool | 877 |
| 7.300.2.3 blt_char | 877 |
| 7.300.2.4 blt_int16s | 877 |
| 7.300.2.5 blt_int16u | 877 |
| 7.300.2.6 blt_int32s | 878 |
| 7.300.2.7 blt_int32u | 878 |
| 7.300.2.8 blt_int8s | 878 |
| 7.300.2.9 blt_int8u | 878 |
| 7.301 types.h File Reference | 878 |
| 7.301.1 Detailed Description | 879 |
| 7.301.2 Typedef Documentation | 879 |
| 7.301.2.1 blt_addr | 879 |
| 7.301.2.2 blt_bool | 879 |
| 7.301.2.3 blt_char | 879 |
| 7.301.2.4 blt_int16s | 879 |
| 7.301.2.5 blt_int16u | 880 |
| 7.301.2.6 blt_int32s | 880 |
| 7.301.2.7 blt_int32u | 880 |
| 7.301.2.8 blt_int8s | 880 |
| 7.301.2.9 blt_int8u | 880 |
| 7.302 types.h File Reference | 880 |
| 7.302.1 Detailed Description | 881 |
| 7.302.2 Typedef Documentation | 881 |
| 7.302.2.1 blt_addr | 881 |
| 7.302.2.2 blt_bool | 881 |
| 7.302.2.3 blt_char | 881 |
| 7.302.2.4 blt_int16s | 881 |
| 7.302.2.5 blt_int16u | 882 |
| 7.302.2.6 blt_int32s | 882 |
| 7.302.2.7 blt_int32u | 882 |
| 7.302.2.8 blt_int8s | 882 |

| | |
|--|-----|
| 7.302.2.9 blt_int8u | 882 |
| 7.303 types.h File Reference | 882 |
| 7.303.1 Detailed Description | 883 |
| 7.303.2 Typedef Documentation | 883 |
| 7.303.2.1 blt_addr | 883 |
| 7.303.2.2 blt_bool | 883 |
| 7.303.2.3 blt_char | 883 |
| 7.303.2.4 blt_int16s | 883 |
| 7.303.2.5 blt_int16u | 884 |
| 7.303.2.6 blt_int32s | 884 |
| 7.303.2.7 blt_int32u | 884 |
| 7.303.2.8 blt_int8s | 884 |
| 7.303.2.9 blt_int8u | 884 |
| 7.304 types.h File Reference | 884 |
| 7.304.1 Detailed Description | 885 |
| 7.304.2 Typedef Documentation | 885 |
| 7.304.2.1 blt_addr | 885 |
| 7.304.2.2 blt_bool | 885 |
| 7.304.2.3 blt_char | 885 |
| 7.304.2.4 blt_int16s | 885 |
| 7.304.2.5 blt_int16u | 886 |
| 7.304.2.6 blt_int32s | 886 |
| 7.304.2.7 blt_int32u | 886 |
| 7.304.2.8 blt_int8s | 886 |
| 7.304.2.9 blt_int8u | 886 |
| 7.305 types.h File Reference | 886 |
| 7.305.1 Detailed Description | 887 |
| 7.305.2 Typedef Documentation | 887 |
| 7.305.2.1 blt_addr | 887 |
| 7.305.2.2 blt_bool | 887 |
| 7.305.2.3 blt_char | 887 |
| 7.305.2.4 blt_int16s | 887 |
| 7.305.2.5 blt_int16u | 888 |
| 7.305.2.6 blt_int32s | 888 |
| 7.305.2.7 blt_int32u | 888 |
| 7.305.2.8 blt_int8s | 888 |
| 7.305.2.9 blt_int8u | 888 |
| 7.306 usb.c File Reference | 888 |
| 7.306.1 Detailed Description | 890 |
| 7.306.2 Function Documentation | 890 |
| 7.306.2.1 UsbFifoMgrCreate() | 890 |
| 7.306.2.2 UsbFifoMgrInit() | 890 |

| | |
|------------------------------------|-----|
| 7.306.2.3 UsbFifoMgrRead() | 891 |
| 7.306.2.4 UsbFifoMgrScan() | 891 |
| 7.306.2.5 UsbFifoMgrWrite() | 892 |
| 7.306.2.6 UsbFree() | 892 |
| 7.306.2.7 UsbInit() | 892 |
| 7.306.2.8 UsbReceiveByte() | 892 |
| 7.306.2.9 UsbReceivePacket() | 893 |
| 7.306.2.10 UsbReceivePipeBulkOUT() | 893 |
| 7.306.2.11 UsbTransmitByte() | 893 |
| 7.306.2.12 UsbTransmitPacket() | 894 |
| 7.306.2.13 UsbTransmitPipeBulkIN() | 894 |
| 7.307 usb.c File Reference | 895 |
| 7.307.1 Detailed Description | 895 |
| 7.307.2 Function Documentation | 895 |
| 7.307.2.1 tud_suspend_cb() | 896 |
| 7.307.2.2 UsbFree() | 896 |
| 7.307.2.3 UsbInit() | 896 |
| 7.307.2.4 UsbReceiveByte() | 896 |
| 7.307.2.5 UsbReceivePacket() | 897 |
| 7.307.2.6 UsbTransmitPacket() | 897 |
| 7.308 usb.c File Reference | 898 |
| 7.308.1 Detailed Description | 899 |
| 7.308.2 Function Documentation | 900 |
| 7.308.2.1 UsbFifoMgrCreate() | 900 |
| 7.308.2.2 UsbFifoMgrInit() | 900 |
| 7.308.2.3 UsbFifoMgrRead() | 900 |
| 7.308.2.4 UsbFifoMgrScan() | 901 |
| 7.308.2.5 UsbFifoMgrWrite() | 901 |
| 7.308.2.6 UsbFree() | 902 |
| 7.308.2.7 UsbInit() | 902 |
| 7.308.2.8 UsbReceiveByte() | 902 |
| 7.308.2.9 UsbReceivePacket() | 903 |
| 7.308.2.10 UsbReceivePipeBulkOUT() | 903 |
| 7.308.2.11 UsbTransmitByte() | 903 |
| 7.308.2.12 UsbTransmitPacket() | 904 |
| 7.308.2.13 UsbTransmitPipeBulkIN() | 904 |
| 7.309 usb.c File Reference | 904 |
| 7.309.1 Detailed Description | 906 |
| 7.309.2 Function Documentation | 906 |
| 7.309.2.1 UsbFifoMgrCreate() | 906 |
| 7.309.2.2 UsbFifoMgrInit() | 907 |
| 7.309.2.3 UsbFifoMgrRead() | 907 |

| | |
|------------------------------------|-----|
| 7.309.2.4 UsbFifoMgrScan() | 907 |
| 7.309.2.5 UsbFifoMgrWrite() | 908 |
| 7.309.2.6 UsbFree() | 908 |
| 7.309.2.7 UsbInit() | 908 |
| 7.309.2.8 UsbReceiveByte() | 909 |
| 7.309.2.9 UsbReceivePacket() | 909 |
| 7.309.2.10 UsbReceivePipeBulkOUT() | 909 |
| 7.309.2.11 UsbTransmitByte() | 910 |
| 7.309.2.12 UsbTransmitPacket() | 911 |
| 7.309.2.13 UsbTransmitPipeBulkIN() | 911 |
| 7.310 usb.c File Reference | 912 |
| 7.310.1 Detailed Description | 913 |
| 7.310.2 Function Documentation | 913 |
| 7.310.2.1 UsbFifoMgrCreate() | 913 |
| 7.310.2.2 UsbFifoMgrInit() | 914 |
| 7.310.2.3 UsbFifoMgrRead() | 914 |
| 7.310.2.4 UsbFifoMgrScan() | 915 |
| 7.310.2.5 UsbFifoMgrWrite() | 915 |
| 7.310.2.6 UsbFree() | 915 |
| 7.310.2.7 UsbInit() | 916 |
| 7.310.2.8 UsbReceiveByte() | 916 |
| 7.310.2.9 UsbReceivePacket() | 916 |
| 7.310.2.10 UsbReceivePipeBulkOUT() | 917 |
| 7.310.2.11 UsbTransmitByte() | 917 |
| 7.310.2.12 UsbTransmitPacket() | 917 |
| 7.310.2.13 UsbTransmitPipeBulkIN() | 918 |
| 7.311 usb.c File Reference | 918 |
| 7.311.1 Detailed Description | 919 |
| 7.311.2 Function Documentation | 920 |
| 7.311.2.1 UsbFifoMgrCreate() | 920 |
| 7.311.2.2 UsbFifoMgrInit() | 920 |
| 7.311.2.3 UsbFifoMgrRead() | 920 |
| 7.311.2.4 UsbFifoMgrScan() | 921 |
| 7.311.2.5 UsbFifoMgrWrite() | 921 |
| 7.311.2.6 UsbFree() | 922 |
| 7.311.2.7 UsbInit() | 922 |
| 7.311.2.8 UsbReceiveByte() | 922 |
| 7.311.2.9 UsbReceivePacket() | 923 |
| 7.311.2.10 UsbReceivePipeBulkOUT() | 923 |
| 7.311.2.11 UsbTransmitByte() | 923 |
| 7.311.2.12 UsbTransmitPacket() | 924 |
| 7.311.2.13 UsbTransmitPipeBulkIN() | 924 |

| | |
|--|-----|
| 7.312 usb.c File Reference | 924 |
| 7.312.1 Detailed Description | 926 |
| 7.312.2 Function Documentation | 926 |
| 7.312.2.1 UsbFifoMgrCreate() | 926 |
| 7.312.2.2 UsbFifoMgrInit() | 927 |
| 7.312.2.3 UsbFifoMgrRead() | 927 |
| 7.312.2.4 UsbFifoMgrScan() | 927 |
| 7.312.2.5 UsbFifoMgrWrite() | 928 |
| 7.312.2.6 UsbFree() | 928 |
| 7.312.2.7 UsbInit() | 928 |
| 7.312.2.8 UsbReceiveByte() | 929 |
| 7.312.2.9 UsbReceivePacket() | 929 |
| 7.312.2.10 UsbReceivePipeBulkOUT() | 929 |
| 7.312.2.11 UsbTransmitByte() | 930 |
| 7.312.2.12 UsbTransmitPacket() | 931 |
| 7.312.2.13 UsbTransmitPipeBulkIN() | 931 |
| 7.313 usb.c File Reference | 932 |
| 7.313.1 Detailed Description | 933 |
| 7.313.2 Function Documentation | 933 |
| 7.313.2.1 UsbFifoMgrCreate() | 933 |
| 7.313.2.2 UsbFifoMgrInit() | 934 |
| 7.313.2.3 UsbFifoMgrRead() | 934 |
| 7.313.2.4 UsbFifoMgrScan() | 935 |
| 7.313.2.5 UsbFifoMgrWrite() | 935 |
| 7.313.2.6 UsbFree() | 935 |
| 7.313.2.7 UsbInit() | 936 |
| 7.313.2.8 UsbReceiveByte() | 936 |
| 7.313.2.9 UsbReceivePacket() | 936 |
| 7.313.2.10 UsbReceivePipeBulkOUT() | 937 |
| 7.313.2.11 UsbTransmitByte() | 937 |
| 7.313.2.12 UsbTransmitPacket() | 937 |
| 7.313.2.13 UsbTransmitPipeBulkIN() | 938 |
| 7.314 usb.c File Reference | 938 |
| 7.314.1 Detailed Description | 939 |
| 7.314.2 Function Documentation | 940 |
| 7.314.2.1 UsbFifoMgrCreate() | 940 |
| 7.314.2.2 UsbFifoMgrInit() | 940 |
| 7.314.2.3 UsbFifoMgrRead() | 940 |
| 7.314.2.4 UsbFifoMgrScan() | 941 |
| 7.314.2.5 UsbFifoMgrWrite() | 941 |
| 7.314.2.6 UsbFree() | 942 |
| 7.314.2.7 UsbInit() | 942 |

| | |
|------------------------------------|-----|
| 7.314.2.8 UsbReceiveByte() | 942 |
| 7.314.2.9 UsbReceivePacket() | 943 |
| 7.314.2.10 UsbReceivePipeBulkOUT() | 943 |
| 7.314.2.11 UsbTransmitByte() | 943 |
| 7.314.2.12 UsbTransmitPacket() | 944 |
| 7.314.2.13 UsbTransmitPipeBulkIN() | 944 |
| 7.315 usb.c File Reference | 944 |
| 7.315.1 Detailed Description | 945 |
| 7.315.2 Function Documentation | 945 |
| 7.315.2.1 tud_suspend_cb() | 945 |
| 7.315.2.2 UsbFree() | 946 |
| 7.315.2.3 UsbInit() | 946 |
| 7.315.2.4 UsbReceiveByte() | 946 |
| 7.315.2.5 UsbReceivePacket() | 946 |
| 7.315.2.6 UsbTransmitPacket() | 947 |
| 7.316 usb.c File Reference | 947 |
| 7.316.1 Detailed Description | 948 |
| 7.316.2 Function Documentation | 948 |
| 7.316.2.1 tud_suspend_cb() | 948 |
| 7.316.2.2 UsbFree() | 949 |
| 7.316.2.3 UsbInit() | 949 |
| 7.316.2.4 UsbReceiveByte() | 949 |
| 7.316.2.5 UsbReceivePacket() | 949 |
| 7.316.2.6 UsbTransmitPacket() | 950 |
| 7.317 usb.c File Reference | 950 |
| 7.317.1 Detailed Description | 952 |
| 7.317.2 Function Documentation | 952 |
| 7.317.2.1 UsbFifoMgrCreate() | 952 |
| 7.317.2.2 UsbFifoMgrInit() | 953 |
| 7.317.2.3 UsbFifoMgrRead() | 953 |
| 7.317.2.4 UsbFifoMgrScan() | 953 |
| 7.317.2.5 UsbFifoMgrWrite() | 954 |
| 7.317.2.6 UsbFree() | 954 |
| 7.317.2.7 UsbInit() | 954 |
| 7.317.2.8 UsbReceiveByte() | 955 |
| 7.317.2.9 UsbReceivePacket() | 955 |
| 7.317.2.10 UsbReceivePipeBulkOUT() | 955 |
| 7.317.2.11 UsbTransmitByte() | 956 |
| 7.317.2.12 UsbTransmitPacket() | 957 |
| 7.317.2.13 UsbTransmitPipeBulkIN() | 957 |
| 7.318 usb.c File Reference | 958 |
| 7.318.1 Detailed Description | 959 |

| | |
|--|-----|
| 7.318.2 Function Documentation | 959 |
| 7.318.2.1 UsbFifoMgrCreate() | 959 |
| 7.318.2.2 UsbFifoMgrInit() | 960 |
| 7.318.2.3 UsbFifoMgrRead() | 960 |
| 7.318.2.4 UsbFifoMgrScan() | 961 |
| 7.318.2.5 UsbFifoMgrWrite() | 961 |
| 7.318.2.6 UsbFree() | 961 |
| 7.318.2.7 UsbInit() | 962 |
| 7.318.2.8 UsbReceiveByte() | 962 |
| 7.318.2.9 UsbReceivePacket() | 962 |
| 7.318.2.10 UsbReceivePipeBulkOUT() | 963 |
| 7.318.2.11 UsbTransmitByte() | 963 |
| 7.318.2.12 UsbTransmitPacket() | 963 |
| 7.318.2.13 UsbTransmitPipeBulkIN() | 964 |
| 7.319 asserts.c File Reference | 964 |
| 7.319.1 Detailed Description | 964 |
| 7.319.2 Function Documentation | 964 |
| 7.319.2.1 AssertFailure() | 964 |
| 7.320 asserts.h File Reference | 965 |
| 7.320.1 Detailed Description | 965 |
| 7.320.2 Function Documentation | 965 |
| 7.320.2.1 AssertFailure() | 965 |
| 7.321 backdoor.c File Reference | 966 |
| 7.321.1 Detailed Description | 966 |
| 7.321.2 Function Documentation | 966 |
| 7.321.2.1 BackDoorCheck() | 967 |
| 7.321.2.2 BackDoorInit() | 967 |
| 7.322 backdoor.h File Reference | 967 |
| 7.322.1 Detailed Description | 968 |
| 7.322.2 Function Documentation | 968 |
| 7.322.2.1 BackDoorCheck() | 968 |
| 7.322.2.2 BackDoorInit() | 968 |
| 7.323 boot.c File Reference | 969 |
| 7.323.1 Detailed Description | 969 |
| 7.323.2 Function Documentation | 969 |
| 7.323.2.1 BootInit() | 969 |
| 7.323.2.2 BootTask() | 970 |
| 7.324 boot.h File Reference | 970 |
| 7.324.1 Detailed Description | 971 |
| 7.324.2 Function Documentation | 971 |
| 7.324.2.1 BootInit() | 971 |
| 7.324.2.2 BootTask() | 971 |

| | |
|---|-----|
| 7.325 can.h File Reference | 971 |
| 7.325.1 Detailed Description | 972 |
| 7.325.2 Function Documentation | 972 |
| 7.325.2.1 CanInit() | 972 |
| 7.325.2.2 CanReceivePacket() | 972 |
| 7.325.2.3 CanTransmitPacket() | 973 |
| 7.326 com.c File Reference | 973 |
| 7.326.1 Detailed Description | 974 |
| 7.326.2 Function Documentation | 974 |
| 7.326.2.1 ComFree() | 974 |
| 7.326.2.2 ComGetActiveInterfaceMaxRxLen() | 975 |
| 7.326.2.3 ComGetActiveInterfaceMaxTxLen() | 975 |
| 7.326.2.4 ComInit() | 975 |
| 7.326.2.5 ComIsConnected() | 975 |
| 7.326.2.6 ComTask() | 976 |
| 7.326.2.7 ComTransmitPacket() | 976 |
| 7.327 com.h File Reference | 976 |
| 7.327.1 Detailed Description | 978 |
| 7.327.2 Enumeration Type Documentation | 978 |
| 7.327.2.1 tComInterfaceId | 978 |
| 7.327.3 Function Documentation | 979 |
| 7.327.3.1 ComFree() | 979 |
| 7.327.3.2 ComGetActiveInterfaceMaxRxLen() | 979 |
| 7.327.3.3 ComGetActiveInterfaceMaxTxLen() | 979 |
| 7.327.3.4 ComInit() | 980 |
| 7.327.3.5 ComIsConnected() | 980 |
| 7.327.3.6 ComTask() | 980 |
| 7.327.3.7 ComTransmitPacket() | 980 |
| 7.328 cop.c File Reference | 981 |
| 7.328.1 Detailed Description | 981 |
| 7.328.2 Function Documentation | 981 |
| 7.328.2.1 CopInit() | 982 |
| 7.328.2.2 CopService() | 982 |
| 7.329 cop.h File Reference | 982 |
| 7.329.1 Detailed Description | 983 |
| 7.329.2 Function Documentation | 983 |
| 7.329.2.1 CopInit() | 983 |
| 7.329.2.2 CopService() | 983 |
| 7.330 cpu.h File Reference | 983 |
| 7.330.1 Detailed Description | 984 |
| 7.330.2 Function Documentation | 984 |
| 7.330.2.1 CpuInit() | 984 |

| | |
|---|------|
| 7.330.2.2 CpuIrqDisable() | 984 |
| 7.330.2.3 CpuIrqEnable() | 985 |
| 7.330.2.4 CpuMemCopy() | 985 |
| 7.330.2.5 CpuMemSet() | 985 |
| 7.330.2.6 CpuStartUserProgram() | 986 |
| 7.331 file.c File Reference | 986 |
| 7.331.1 Detailed Description | 988 |
| 7.331.2 Enumeration Type Documentation | 988 |
| 7.331.2.1 tFirmwareUpdateState | 988 |
| 7.331.3 Function Documentation | 988 |
| 7.331.3.1 FileHandleFirmwareUpdateRequest() | 988 |
| 7.331.3.2 FileInit() | 989 |
| 7.331.3.3 FileIsIdle() | 989 |
| 7.331.3.4 FileLibByteNibbleToChar() | 989 |
| 7.331.3.5 FileLibByteToHexString() | 990 |
| 7.331.3.6 FileLibHexStringToByte() | 990 |
| 7.331.3.7 FileLibLongToIntString() | 990 |
| 7.331.3.8 FileSrecGetLineType() | 991 |
| 7.331.3.9 FileSrecParseLine() | 991 |
| 7.331.3.10 FileSrecVerifyChecksum() | 992 |
| 7.331.3.11 FileTask() | 992 |
| 7.332 file.h File Reference | 992 |
| 7.332.1 Detailed Description | 994 |
| 7.332.2 Enumeration Type Documentation | 994 |
| 7.332.2.1 tSrecLineType | 994 |
| 7.332.3 Function Documentation | 995 |
| 7.332.3.1 FileHandleFirmwareUpdateRequest() | 995 |
| 7.332.3.2 FileInit() | 995 |
| 7.332.3.3 FileIsIdle() | 995 |
| 7.332.3.4 FileSrecGetLineType() | 995 |
| 7.332.3.5 FileSrecParseLine() | 996 |
| 7.332.3.6 FileSrecVerifyChecksum() | 996 |
| 7.332.3.7 FileTask() | 997 |
| 7.333 flash_ecc.c File Reference | 997 |
| 7.333.1 Detailed Description | 1000 |
| 7.333.2 Function Documentation | 1000 |
| 7.333.2.1 FlashAddToBlock() | 1000 |
| 7.333.2.2 FlashDone() | 1001 |
| 7.333.2.3 FlashErase() | 1001 |
| 7.333.2.4 FlashExecuteCommand() | 1001 |
| 7.333.2.5 FlashGetGlobalAddrByte() | 1002 |
| 7.333.2.6 FlashGetPhysAddr() | 1002 |

| | |
|--|------|
| 7.333.2.7 FlashGetPhysPage() | 1002 |
| 7.333.2.8 Flash GetUserProgBaseAddress() | 1003 |
| 7.333.2.9 FlashInit() | 1003 |
| 7.333.2.10 FlashInitBlock() | 1003 |
| 7.333.2.11 FlashOperate() | 1004 |
| 7.333.2.12 FlashSwitchBlock() | 1004 |
| 7.333.2.13 FlashVerifyChecksum() | 1005 |
| 7.333.2.14 FlashWrite() | 1005 |
| 7.333.2.15 FlashWriteBlock() | 1005 |
| 7.333.2.16 FlashWriteChecksum() | 1006 |
| 7.333.3 Variable Documentation | 1006 |
| 7.333.3.1 blockInfo | 1006 |
| 7.333.3.2 bootBlockInfo | 1007 |
| 7.333.3.3 flashExecCmd | 1007 |
| 7.333.3.4 flashLayout | 1008 |
| 7.334 mb.c File Reference | 1008 |
| 7.334.1 Detailed Description | 1008 |
| 7.335 mb.h File Reference | 1008 |
| 7.335.1 Detailed Description | 1009 |
| 7.336 net.c File Reference | 1009 |
| 7.336.1 Detailed Description | 1010 |
| 7.336.2 Function Documentation | 1010 |
| 7.336.2.1 NetApp() | 1010 |
| 7.336.2.2 NetInit() | 1010 |
| 7.336.2.3 NetReceivePacket() | 1010 |
| 7.336.2.4 NetServerTask() | 1011 |
| 7.336.2.5 NetTransmitPacket() | 1011 |
| 7.337 net.h File Reference | 1012 |
| 7.337.1 Detailed Description | 1012 |
| 7.337.2 Function Documentation | 1012 |
| 7.337.2.1 NetApp() | 1013 |
| 7.337.2.2 NetInit() | 1013 |
| 7.337.2.3 NetReceivePacket() | 1013 |
| 7.337.2.4 NetTransmitPacket() | 1014 |
| 7.338 nvm.h File Reference | 1014 |
| 7.338.1 Detailed Description | 1015 |
| 7.338.2 Function Documentation | 1015 |
| 7.338.2.1 NvmDone() | 1015 |
| 7.338.2.2 NvmErase() | 1015 |
| 7.338.2.3 Nvm GetUserProgBaseAddress() | 1016 |
| 7.338.2.4 NvmInit() | 1016 |
| 7.338.2.5 NvmVerifyChecksum() | 1016 |

| | |
|-------------------------------------|------|
| 7.338.2.6 NvmWrite() | 1016 |
| 7.339 plausibility.h File Reference | 1017 |
| 7.339.1 Detailed Description | 1017 |
| 7.340 rs232.h File Reference | 1017 |
| 7.340.1 Detailed Description | 1017 |
| 7.341 timer.h File Reference | 1017 |
| 7.341.1 Detailed Description | 1018 |
| 7.341.2 Function Documentation | 1018 |
| 7.341.2.1 TimerGet() | 1018 |
| 7.341.2.2 TimerInit() | 1019 |
| 7.341.2.3 TimerReset() | 1020 |
| 7.341.2.4 TimerUpdate() | 1020 |
| 7.342 ram_func.h File Reference | 1020 |
| 7.342.1 Detailed Description | 1020 |
| 7.343 ram_func.h File Reference | 1021 |
| 7.343.1 Detailed Description | 1021 |
| 7.344 usb.h File Reference | 1021 |
| 7.344.1 Detailed Description | 1022 |
| 7.344.2 Function Documentation | 1022 |
| 7.344.2.1 UsbFree() | 1022 |
| 7.344.2.2 UsbInit() | 1022 |
| 7.344.2.3 UsbReceivePacket() | 1022 |
| 7.344.2.4 UsbTransmitPacket() | 1023 |
| 7.345 xcp.c File Reference | 1023 |
| 7.345.1 Detailed Description | 1025 |
| 7.345.2 Function Documentation | 1025 |
| 7.345.2.1 XcpCmdBuildCheckSum() | 1025 |
| 7.345.2.2 XcpCmdConnect() | 1026 |
| 7.345.2.3 XcpCmdDisconnect() | 1026 |
| 7.345.2.4 XcpCmdGetId() | 1026 |
| 7.345.2.5 XcpCmdGetSeed() | 1027 |
| 7.345.2.6 XcpCmdGetStatus() | 1027 |
| 7.345.2.7 XcpCmdProgram() | 1028 |
| 7.345.2.8 XcpCmdProgramClear() | 1028 |
| 7.345.2.9 XcpCmdProgramMax() | 1028 |
| 7.345.2.10 XcpCmdProgramPrepare() | 1029 |
| 7.345.2.11 XcpCmdProgramReset() | 1029 |
| 7.345.2.12 XcpCmdProgramStart() | 1029 |
| 7.345.2.13 XcpCmdSetMta() | 1030 |
| 7.345.2.14 XcpCmdShortUpload() | 1030 |
| 7.345.2.15 XcpCmdSynch() | 1031 |
| 7.345.2.16 XcpCmdUnlock() | 1031 |

| | |
|--|-------------|
| 7.345.2.17 XcpCmdUpload() | 1031 |
| 7.345.2.18 XcpComputeChecksum() | 1032 |
| 7.345.2.19 XcpGetOrderedLong() | 1032 |
| 7.345.2.20 XcpGetSeed() | 1033 |
| 7.345.2.21 XcpInit() | 1033 |
| 7.345.2.22 XcpIsConnected() | 1033 |
| 7.345.2.23 XcpPacketReceived() | 1033 |
| 7.345.2.24 XcpPacketTransmitted() | 1034 |
| 7.345.2.25 XcpProtectResources() | 1034 |
| 7.345.2.26 XcpSetCtoError() | 1034 |
| 7.345.2.27 XcpSetOrderedLong() | 1035 |
| 7.345.2.28 XcpTransmitPacket() | 1035 |
| 7.345.2.29 XcpVerifyKey() | 1036 |
| 7.346 xcp.h File Reference | 1036 |
| 7.346.1 Detailed Description | 1039 |
| 7.346.2 Macro Definition Documentation | 1039 |
| 7.346.2.1 XCP_PACKET RECEIVED HOOK EN | 1039 |
| 7.346.3 Function Documentation | 1040 |
| 7.346.3.1 XcpInit() | 1040 |
| 7.346.3.2 XcpIsConnected() | 1040 |
| 7.346.3.3 XcpPacketReceived() | 1040 |
| 7.346.3.4 XcpPacketTransmitted() | 1041 |
| Index | 1043 |

Chapter 1

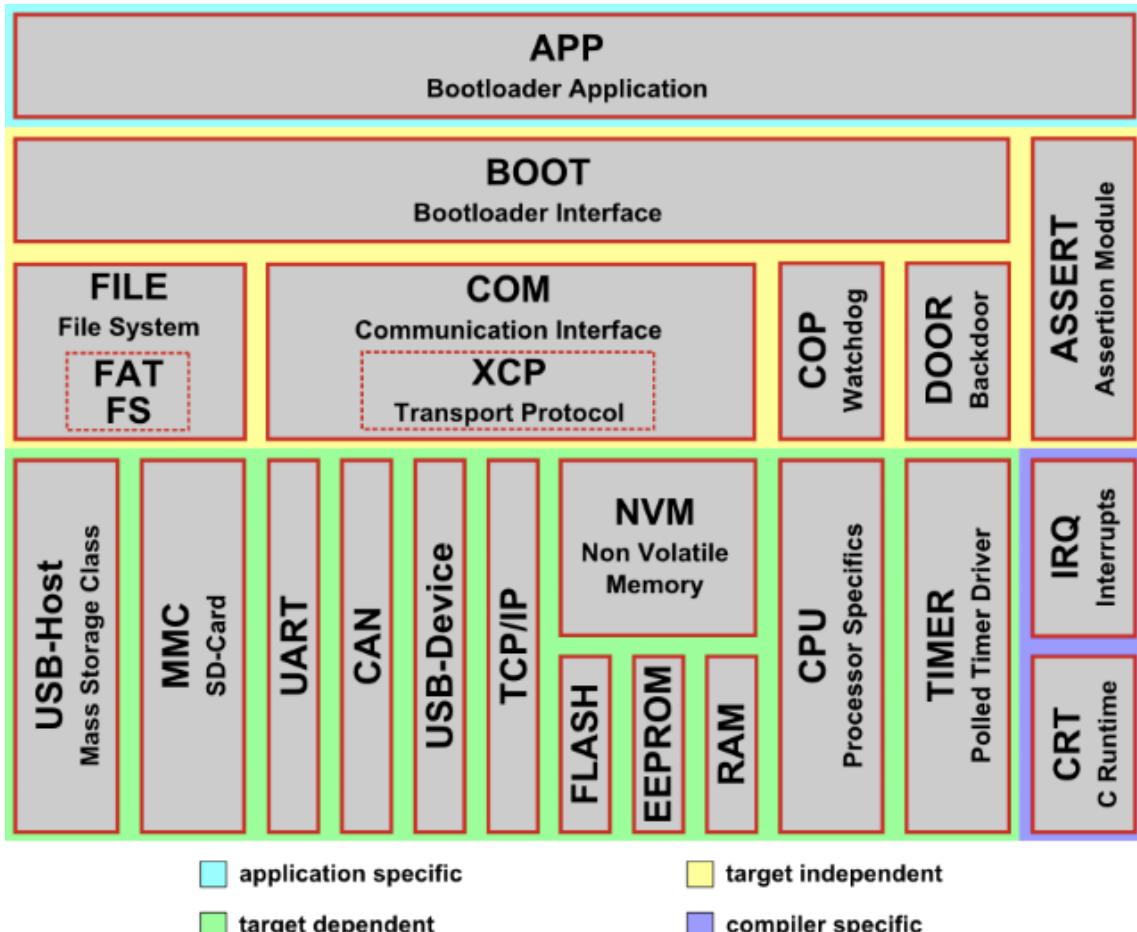
OpenBLT Firmware Documentation

1.1 Introduction

This documentation covers the OpenBLT (Open source BootLoader Tool) firmware. With OpenBLT you can make software updates through an on-chip communication interface (RS232, CAN, TCP/IP, USB, Modbus-RTU, etc.), without the need of specialized debugger hardware.

1.2 Software Architecture

The software program's architecture is divided into 4 major categories, namely the application code (App), target independent code (Core), target dependent code (Target), and compiler specific code (Comp).



To configure and fine-tune the bootloader for integration in your product, all you have to do is take the demo bootloader project for the microcontroller and compiler you are using, and (optionally) modify just the application code (App) to fit your needs. This typically involves changing the configuration header file (`blt_conf.h`) and the implementation of the hook functions (`hooks.c`).

For more in-depth information behind the design of the OpenBLT bootloader, you can visit: <https://www.feaser.com/openblt/doku.php?id=manual:design>.

1.3 Copyright and Licensing

C O P Y R I G H T

Copyright (c) by Feaser 2011-2025 <http://www.feaser.com> All rights reserved

L I C E N S E

This file is part of OpenBLT. OpenBLT is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

OpenBLT is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You have received a copy of the GNU General Public License along with OpenBLT. It should be located in ".\Doc\license.html". If not, contact Feaser to obtain a copy.

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

| | |
|--|----|
| Bootloader Core | 44 |
| Bootloader Ports | 47 |
| Target Port Template | 24 |
| CAN driver of a port | 21 |
| CPU driver of a port | 21 |
| Flash driver of a port | 22 |
| Compiler specifics of a port | 22 |
| Modbus RTU driver of a port | 23 |
| Non-volatile memory driver of a port | 23 |
| RS232 UART driver of a port | 23 |
| Timer driver of a port | 24 |
| Type definitions of a port | 25 |
| USB driver of a port | 25 |
| Target ARMCM0 S32K11 | 26 |
| Target ARMCM0 STM32C0 | 26 |
| Target ARMCM0 STM32F0 | 27 |
| Target ARMCM0 STM32G0 | 28 |
| Target ARMCM0 STM32L0 | 29 |
| Target ARMCM0 XMC1 | 29 |
| Target ARMCM33 STM32H5 | 30 |
| Target ARMCM33 STM32L5 | 31 |
| Target ARMCM33 STM32U5 | 32 |
| Target ARMCM3 EFM32 | 33 |
| Target ARMCM3 LM3S | 33 |
| Target ARMCM3 STM32F1 | 34 |
| Target ARMCM3 STM32F2 | 35 |
| Target ARMCM3 STM32L1 | 36 |
| Target ARMCM4 S32K14 | 37 |
| Target ARMCM4 STM32F3 | 37 |
| Target ARMCM4 STM32F4 | 38 |
| Target ARMCM4 STM32G4 | 39 |
| Target ARMCM4 STM32L4 | 40 |
| Target ARMCM4 TM4C | 41 |
| Target ARMCM4 XMC4 | 42 |
| Target ARMCM7 STM32F7 | 43 |

| | |
|---------------------------------|----|
| Target ARCMC7 STM32H7 | 43 |
| Target HCS12 | 46 |
| Target TRICORE TC2 | 48 |
| Target TRICORE TC3 | 49 |

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

| | | |
|--|--|----|
| tCanBusTiming | Structure type for grouping CAN bus timing related information | 51 |
| tCanRegs | Structure type with the layout of the CAN related control registers | 53 |
| tCanRxMsgSlot | Structure type with the layout of a CAN reception message slot | 59 |
| tCanTxMsgSlot | Structure type with the layout of a CAN transmit message slot | 60 |
| tFatFsObjects | Structure type for grouping FATFS related objects used by this module | 62 |
| tFifoCtrl | Structure type for fifo control | 63 |
| tFifoPipe | Structure type for a fifo pipe | 65 |
| tFileEraseInfo | Structure type with information for the memory erase opeartion | 66 |
| tFlashBlockInfo | Structure type for grouping flash block information | 67 |
| tFlashPrescalerSysclockMapping | Mapping table for finding the corect flash clock divider prescaler | 68 |
| tFlashRegs | Structure type for the flash control registers | 69 |
| tFlashSector | Flash sector descriptor type | 73 |
| tSrecLineParseObject | Structure type for grouping the parsing results of an S-record line | 75 |
| tTimerRegs | Structure type with the layout of the timer related control registers | 76 |
| tXcpInfo | Structure type for grouping XCP internal module information | 79 |
| uip_tcp_appstate_t | Define the <code>uip_tcp_appstate_t</code> datatype. This is the state of our tcp/ip application, and the memory required for this state is allocated together with each TCP connection. One application state for each TCP connection | 81 |

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

| | | |
|---------------------------------------|--|-----|
| _template/can.c | Bootloader CAN communication interface source file | 83 |
| ARMCM0_S32K11/can.c | Bootloader CAN communication interface source file | 86 |
| ARMCM0_STM32F0/can.c | Bootloader CAN communication interface source file | 91 |
| ARMCM0_STM32G0/can.c | Bootloader CAN communication interface source file | 95 |
| ARMCM0_XMC1/can.c | Bootloader CAN communication interface source file | 98 |
| ARMCM33_STM32H5/can.c | Bootloader CAN communication interface source file | 100 |
| ARMCM33_STM32L5/can.c | Bootloader CAN communication interface source file | 104 |
| ARMCM33_STM32U5/can.c | Bootloader CAN communication interface source file | 107 |
| ARMCM3_LM3S/can.c | Bootloader CAN communication interface source file | 111 |
| ARMCM3_STM32F1/can.c | Bootloader CAN communication interface source file | 113 |
| ARMCM3_STM32F2/can.c | Bootloader CAN communication interface source file | 117 |
| ARMCM4_S32K14/can.c | Bootloader CAN communication interface source file | 120 |
| ARMCM4_STM32F3/can.c | Bootloader CAN communication interface source file | 125 |
| ARMCM4_STM32F4/can.c | Bootloader CAN communication interface source file | 129 |
| ARMCM4_STM32G4/can.c | Bootloader CAN communication interface source file | 132 |
| ARMCM4_STM32L4/can.c | Bootloader CAN communication interface source file | 136 |
| ARMCM4_XMC4/can.c | Bootloader CAN communication interface source file | 139 |
| ARMCM7_STM32F7/can.c | Bootloader CAN communication interface source file | 141 |

| | |
|--|-----|
| ARMCM7_STM32H7/can.c | |
| Bootloader CAN communication interface source file | 145 |
| HCS12/can.c | |
| Bootloader CAN communication interface source file | 148 |
| TRICORE_TC2/can.c | |
| Bootloader CAN communication interface source file | 152 |
| TRICORE_TC3/can.c | |
| Bootloader CAN communication interface source file | 156 |
| _template/cpu.c | |
| Bootloader cpu module source file | 159 |
| ARMCM0_S32K11/cpu.c | |
| Bootloader cpu module source file | 162 |
| ARMCM0_STM32C0/cpu.c | |
| Bootloader cpu module source file | 164 |
| ARMCM0_STM32F0/cpu.c | |
| Bootloader cpu module source file | 166 |
| ARMCM0_STM32G0/cpu.c | |
| Bootloader cpu module source file | 169 |
| ARMCM0_STM32L0/cpu.c | |
| Bootloader cpu module source file | 171 |
| ARMCM0_XMC1/cpu.c | |
| Bootloader cpu module source file | 174 |
| ARMCM33_STM32H5/cpu.c | |
| Bootloader cpu module source file | 176 |
| ARMCM33_STM32L5/cpu.c | |
| Bootloader cpu module source file | 178 |
| ARMCM33_STM32U5/cpu.c | |
| Bootloader cpu module source file | 181 |
| ARMCM3_EFM32/cpu.c | |
| Bootloader cpu module source file | 183 |
| ARMCM3_LM3S/cpu.c | |
| Bootloader cpu module source file | 185 |
| ARMCM3_STM32F1/cpu.c | |
| Bootloader cpu module source file | 188 |
| ARMCM3_STM32F2/cpu.c | |
| Bootloader cpu module source file | 190 |
| ARMCM3_STM32L1/cpu.c | |
| Bootloader cpu module source file | 192 |
| ARMCM4_S32K14/cpu.c | |
| Bootloader cpu module source file | 195 |
| ARMCM4_STM32F3/cpu.c | |
| Bootloader cpu module source file | 197 |
| ARMCM4_STM32F4/cpu.c | |
| Bootloader cpu module source file | 199 |
| ARMCM4_STM32G4/cpu.c | |
| Bootloader cpu module source file | 202 |
| ARMCM4_STM32L4/cpu.c | |
| Bootloader cpu module source file | 204 |
| ARMCM4_TM4C/cpu.c | |
| Bootloader cpu module source file | 206 |
| ARMCM4_XMC4/cpu.c | |
| Bootloader cpu module source file | 209 |
| ARMCM7_STM32F7/cpu.c | |
| Bootloader cpu module source file | 211 |
| ARMCM7_STM32H7/cpu.c | |
| Bootloader cpu module source file | 213 |
| HCS12/cpu.c | |
| Bootloader cpu module source file | 216 |

| | | |
|-------------------------|-------------------------------------|-----|
| TRICORE_TC2/cpu.c | Bootloader cpu module source file | 218 |
| TRICORE_TC3/cpu.c | Bootloader cpu module source file | 221 |
| _template/flash.c | Bootloader flash driver source file | 224 |
| ARMCM0_S32K11/flash.c | Bootloader flash driver source file | 232 |
| ARMCM0_STM32C0/flash.c | Bootloader flash driver source file | 240 |
| ARMCM0_STM32F0/flash.c | Bootloader flash driver source file | 249 |
| ARMCM0_STM32G0/flash.c | Bootloader flash driver source file | 258 |
| ARMCM0_STM32L0/flash.c | Bootloader flash driver source file | 269 |
| ARMCM0_XMC1/flash.c | Bootloader flash driver source file | 277 |
| ARMCM33_STM32H5/flash.c | Bootloader flash driver source file | 286 |
| ARMCM33_STM32L5/flash.c | Bootloader flash driver source file | 297 |
| ARMCM33_STM32U5/flash.c | Bootloader flash driver source file | 306 |
| ARMCM3_EFM32/flash.c | Bootloader flash driver source file | 315 |
| ARMCM3_LM3S/flash.c | Bootloader flash driver source file | 324 |
| ARMCM3_STM32F1/flash.c | Bootloader flash driver source file | 333 |
| ARMCM3_STM32F2/flash.c | Bootloader flash driver source file | 340 |
| ARMCM3_STM32L1/flash.c | Bootloader flash driver source file | 349 |
| ARMCM4_S32K14/flash.c | Bootloader flash driver source file | 357 |
| ARMCM4_STM32F3/flash.c | Bootloader flash driver source file | 365 |
| ARMCM4_STM32F4/flash.c | Bootloader flash driver source file | 372 |
| ARMCM4_STM32G4/flash.c | Bootloader flash driver source file | 381 |
| ARMCM4_STM32L4/flash.c | Bootloader flash driver source file | 390 |
| ARMCM4_TM4C/flash.c | Bootloader flash driver source file | 398 |
| ARMCM4_XMC4/flash.c | Bootloader flash driver source file | 407 |
| ARMCM7_STM32F7/flash.c | Bootloader flash driver source file | 417 |
| ARMCM7_STM32H7/flash.c | Bootloader flash driver source file | 426 |
| HCS12/flash.c | Bootloader flash driver source file | 434 |
| TRICORE_TC2/flash.c | Bootloader flash driver source file | 445 |
| TRICORE_TC3/flash.c | Bootloader flash driver source file | 455 |

| | | |
|--|-------------------------------------|-----|
| _template/flash.h | Bootloader flash driver header file | 465 |
| ARMCM0_S32K11/flash.h | Bootloader flash driver header file | 468 |
| ARMCM0_STM32C0/flash.h | Bootloader flash driver header file | 472 |
| ARMCM0_STM32F0/flash.h | Bootloader flash driver header file | 476 |
| ARMCM0_STM32G0/flash.h | Bootloader flash driver header file | 480 |
| ARMCM0_STM32L0/flash.h | Bootloader flash driver header file | 484 |
| ARMCM0_XMC1/flash.h | Bootloader flash driver header file | 488 |
| ARMCM33_STM32H5/flash.h | Bootloader flash driver header file | 492 |
| ARMCM33_STM32L5/flash.h | Bootloader flash driver header file | 496 |
| ARMCM33_STM32U5/flash.h | Bootloader flash driver header file | 500 |
| ARMCM3_EFM32/flash.h | Bootloader flash driver header file | 504 |
| ARMCM3_LM3S/flash.h | Bootloader flash driver header file | 508 |
| ARMCM3_STM32F1/flash.h | Bootloader flash driver header file | 512 |
| ARMCM3_STM32F2/flash.h | Bootloader flash driver header file | 516 |
| ARMCM3_STM32L1/flash.h | Bootloader flash driver header file | 520 |
| ARMCM4_S32K14/flash.h | Bootloader flash driver header file | 524 |
| ARMCM4_STM32F3/flash.h | Bootloader flash driver header file | 528 |
| ARMCM4_STM32F4/flash.h | Bootloader flash driver header file | 532 |
| ARMCM4_STM32G4/flash.h | Bootloader flash driver header file | 536 |
| ARMCM4_STM32L4/flash.h | Bootloader flash driver header file | 540 |
| ARMCM4_TM4C/flash.h | Bootloader flash driver header file | 544 |
| ARMCM4_XMC4/flash.h | Bootloader flash driver header file | 548 |
| ARMCM7_STM32F7/flash.h | Bootloader flash driver header file | 552 |
| ARMCM7_STM32H7/flash.h | Bootloader flash driver header file | 556 |
| HCS12/flash.h | Bootloader flash driver header file | 560 |
| TRICORE_TC2/flash.h | Bootloader flash driver header file | 564 |
| TRICORE_TC3/flash.h | Bootloader flash driver header file | 568 |
| _template/GCC/cpu_comp.c | Bootloader cpu module source file | 572 |
| ARMCM0_S32K11/GCC/cpu_comp.c | Bootloader cpu module source file | 573 |

| | | |
|---|-----------------------------------|-----|
| ARMCM0_S32K11/IAR/cpu_comp.c | Bootloader cpu module source file | 575 |
| ARMCM0_STM32C0/GCC/cpu_comp.c | Bootloader cpu module source file | 576 |
| ARMCM0_STM32C0/IAR/cpu_comp.c | Bootloader cpu module source file | 577 |
| ARMCM0_STM32C0/Keil/cpu_comp.c | Bootloader cpu module source file | 578 |
| ARMCM0_STM32F0/GCC/cpu_comp.c | Bootloader cpu module source file | 580 |
| ARMCM0_STM32F0/IAR/cpu_comp.c | Bootloader cpu module source file | 581 |
| ARMCM0_STM32F0/Keil/cpu_comp.c | Bootloader cpu module source file | 582 |
| ARMCM0_STM32G0/GCC/cpu_comp.c | Bootloader cpu module source file | 583 |
| ARMCM0_STM32G0/IAR/cpu_comp.c | Bootloader cpu module source file | 585 |
| ARMCM0_STM32G0/Keil/cpu_comp.c | Bootloader cpu module source file | 586 |
| ARMCM0_STM32L0/GCC/cpu_comp.c | Bootloader cpu module source file | 587 |
| ARMCM0_STM32L0/IAR/cpu_comp.c | Bootloader cpu module source file | 588 |
| ARMCM0_STM32L0/Keil/cpu_comp.c | Bootloader cpu module source file | 590 |
| ARMCM0_XMC1/GCC/cpu_comp.c | Bootloader cpu module source file | 591 |
| ARMCM0_XMC1/IAR/cpu_comp.c | Bootloader cpu module source file | 592 |
| ARMCM0_XMC1/Keil/cpu_comp.c | Bootloader cpu module source file | 593 |
| ARMCM33_STM32H5/GCC/cpu_comp.c | Bootloader cpu module source file | 595 |
| ARMCM33_STM32H5/IAR/cpu_comp.c | Bootloader cpu module source file | 596 |
| ARMCM33_STM32H5/Keil/cpu_comp.c | Bootloader cpu module source file | 597 |
| ARMCM33_STM32L5/GCC/cpu_comp.c | Bootloader cpu module source file | 598 |
| ARMCM33_STM32L5/IAR/cpu_comp.c | Bootloader cpu module source file | 600 |
| ARMCM33_STM32L5/Keil/cpu_comp.c | Bootloader cpu module source file | 601 |
| ARMCM33_STM32U5/GCC/cpu_comp.c | Bootloader cpu module source file | 602 |
| ARMCM33_STM32U5/IAR/cpu_comp.c | Bootloader cpu module source file | 603 |
| ARMCM33_STM32U5/Keil/cpu_comp.c | Bootloader cpu module source file | 605 |
| ARMCM3_EFM32/GCC/cpu_comp.c | Bootloader cpu module source file | 606 |
| ARMCM3_EFM32/IAR/cpu_comp.c | Bootloader cpu module source file | 607 |
| ARMCM3_LM3S/GCC/cpu_comp.c | Bootloader cpu module source file | 608 |
| ARMCM3_LM3S/IAR/cpu_comp.c | Bootloader cpu module source file | 610 |

| | |
|--|-----|
| ARMCM3_STM32F1/GCC/cpu_comp.c | |
| Bootloader cpu module source file | 611 |
| ARMCM3_STM32F1/IAR/cpu_comp.c | |
| Bootloader cpu module source file | 612 |
| ARMCM3_STM32F1/Keil/cpu_comp.c | |
| Bootloader cpu module source file | 613 |
| ARMCM3_STM32F2/GCC/cpu_comp.c | |
| Bootloader cpu module source file | 615 |
| ARMCM3_STM32F2/IAR/cpu_comp.c | |
| Bootloader cpu module source file | 616 |
| ARMCM3_STM32F2/Keil/cpu_comp.c | |
| Bootloader cpu module source file | 617 |
| ARMCM3_STM32L1/GCC/cpu_comp.c | |
| Bootloader cpu module source file | 618 |
| ARMCM3_STM32L1/IAR/cpu_comp.c | |
| Bootloader cpu module source file | 620 |
| ARMCM3_STM32L1/Keil/cpu_comp.c | |
| Bootloader cpu module source file | 621 |
| ARMCM4_S32K14/GCC/cpu_comp.c | |
| Bootloader cpu module source file | 622 |
| ARMCM4_S32K14/IAR/cpu_comp.c | |
| Bootloader cpu module source file | 623 |
| ARMCM4_STM32F3/GCC/cpu_comp.c | |
| Bootloader cpu module source file | 625 |
| ARMCM4_STM32F3/IAR/cpu_comp.c | |
| Bootloader cpu module source file | 626 |
| ARMCM4_STM32F3/Keil/cpu_comp.c | |
| Bootloader cpu module source file | 627 |
| ARMCM4_STM32F4/GCC/cpu_comp.c | |
| Bootloader cpu module source file | 628 |
| ARMCM4_STM32F4/IAR/cpu_comp.c | |
| Bootloader cpu module source file | 630 |
| ARMCM4_STM32F4/Keil/cpu_comp.c | |
| Bootloader cpu module source file | 631 |
| ARMCM4_STM32G4/GCC/cpu_comp.c | |
| Bootloader cpu module source file | 632 |
| ARMCM4_STM32G4/IAR/cpu_comp.c | |
| Bootloader cpu module source file | 633 |
| ARMCM4_STM32G4/Keil/cpu_comp.c | |
| Bootloader cpu module source file | 635 |
| ARMCM4_STM32L4/GCC/cpu_comp.c | |
| Bootloader cpu module source file | 636 |
| ARMCM4_STM32L4/IAR/cpu_comp.c | |
| Bootloader cpu module source file | 637 |
| ARMCM4_STM32L4/Keil/cpu_comp.c | |
| Bootloader cpu module source file | 638 |
| ARMCM4_TM4C/IAR/cpu_comp.c | |
| Bootloader cpu module source file | 640 |
| ARMCM4_XMC4/GCC/cpu_comp.c | |
| Bootloader cpu module source file | 641 |
| ARMCM4_XMC4/IAR/cpu_comp.c | |
| Bootloader cpu module source file | 642 |
| ARMCM4_XMC4/Keil/cpu_comp.c | |
| Bootloader cpu module source file | 643 |
| ARMCM7_STM32F7/GCC/cpu_comp.c | |
| Bootloader cpu module source file | 645 |
| ARMCM7_STM32F7/IAR/cpu_comp.c | |
| Bootloader cpu module source file | 646 |

| | | |
|--|---|-----|
| ARMCM7_STM32F7/Keil/cpu_comp.c | Bootloader cpu module source file | 647 |
| ARMCM7_STM32H7/GCC/cpu_comp.c | Bootloader cpu module source file | 648 |
| ARMCM7_STM32H7/IAR/cpu_comp.c | Bootloader cpu module source file | 650 |
| ARMCM7_STM32H7/Keil/cpu_comp.c | Bootloader cpu module source file | 651 |
| HCS12/CodeWarrior/cpu_comp.c | Bootloader cpu module source file | 652 |
| TRICORE_TC2/Tasking/cpu_comp.c | Bootloader cpu module source file | 653 |
| TRICORE_TC3/Tasking/cpu_comp.c | Bootloader cpu module source file | 655 |
| _template/mbrtu.c | Bootloader Modbus RTU communication interface source file | 656 |
| ARMCM0_S32K11/mbrtu.c | Bootloader Modbus RTU communication interface source file | 656 |
| ARMCM0_STM32C0/mbrtu.c | Bootloader Modbus RTU communication interface source file | 657 |
| ARMCM0_STM32F0/mbrtu.c | Bootloader Modbus RTU communication interface source file | 657 |
| ARMCM0_STM32G0/mbrtu.c | Bootloader Modbus RTU communication interface source file | 658 |
| ARMCM0_STM32L0/mbrtu.c | Bootloader Modbus RTU communication interface source file | 658 |
| ARMCM0_XMC1/mbrtu.c | Bootloader Modbus RTU communication interface source file | 659 |
| ARMCM33_STM32H5/mbrtu.c | Bootloader Modbus RTU communication interface source file | 659 |
| ARMCM33_STM32L5/mbrtu.c | Bootloader Modbus RTU communication interface source file | 660 |
| ARMCM33_STM32U5/mbrtu.c | Bootloader Modbus RTU communication interface source file | 660 |
| ARMCM3_EFM32/mbrtu.c | Bootloader Modbus RTU communication interface source file | 661 |
| ARMCM3_LM3S/mbrtu.c | Bootloader Modbus RTU communication interface source file | 661 |
| ARMCM3_STM32F1/mbrtu.c | Bootloader Modbus RTU communication interface source file | 662 |
| ARMCM3_STM32F2/mbrtu.c | Bootloader Modbus RTU communication interface source file | 662 |
| ARMCM3_STM32L1/mbrtu.c | Bootloader Modbus RTU communication interface source file | 663 |
| ARMCM4_S32K14/mbrtu.c | Bootloader Modbus RTU communication interface source file | 663 |
| ARMCM4_STM32F3/mbrtu.c | Bootloader Modbus RTU communication interface source file | 664 |
| ARMCM4_STM32F4/mbrtu.c | Bootloader Modbus RTU communication interface source file | 664 |
| ARMCM4_STM32G4/mbrtu.c | Bootloader Modbus RTU communication interface source file | 665 |
| ARMCM4_STM32L4/mbrtu.c | Bootloader Modbus RTU communication interface source file | 665 |
| ARMCM4_TM4C/mbrtu.c | Bootloader Modbus RTU communication interface source file | 666 |
| ARMCM4_XMC4/mbrtu.c | Bootloader Modbus RTU communication interface source file | 666 |

| | | |
|--|---|-----|
| ARMCM7_STM32F7/mbrtu.c | Bootloader Modbus RTU communication interface source file | 667 |
| ARMCM7_STM32H7/mbrtu.c | Bootloader Modbus RTU communication interface source file | 667 |
| HCS12/mbrtu.c | Bootloader Modbus RTU communication interface source file | 668 |
| TRICORE_TC2/mbrtu.c | Bootloader Modbus RTU communication interface source file | 668 |
| TRICORE_TC3/mbrtu.c | Bootloader Modbus RTU communication interface source file | 669 |
| _template/nvm.c | Bootloader non-volatile memory driver source file | 669 |
| ARMCM0_S32K11/nvm.c | Bootloader non-volatile memory driver source file | 673 |
| ARMCM0_STM32C0/nvm.c | Bootloader non-volatile memory driver source file | 676 |
| ARMCM0_STM32F0/nvm.c | Bootloader non-volatile memory driver source file | 678 |
| ARMCM0_STM32G0/nvm.c | Bootloader non-volatile memory driver source file | 681 |
| ARMCM0_STM32L0/nvm.c | Bootloader non-volatile memory driver source file | 684 |
| ARMCM0_XMC1/nvm.c | Bootloader non-volatile memory driver source file | 687 |
| ARMCM33_STM32H5/nvm.c | Bootloader non-volatile memory driver source file | 690 |
| ARMCM33_STM32L5/nvm.c | Bootloader non-volatile memory driver source file | 693 |
| ARMCM33_STM32U5/nvm.c | Bootloader non-volatile memory driver source file | 696 |
| ARMCM3_EFM32/nvm.c | Bootloader non-volatile memory driver source file | 699 |
| ARMCM3_LM3S/nvm.c | Bootloader non-volatile memory driver source file | 702 |
| ARMCM3_STM32F1/nvm.c | Bootloader non-volatile memory driver source file | 705 |
| ARMCM3_STM32F2/nvm.c | Bootloader non-volatile memory driver source file | 708 |
| ARMCM3_STM32L1/nvm.c | Bootloader non-volatile memory driver source file | 711 |
| ARMCM4_S32K14/nvm.c | Bootloader non-volatile memory driver source file | 714 |
| ARMCM4_STM32F3/nvm.c | Bootloader non-volatile memory driver source file | 717 |
| ARMCM4_STM32F4/nvm.c | Bootloader non-volatile memory driver source file | 720 |
| ARMCM4_STM32G4/nvm.c | Bootloader non-volatile memory driver source file | 723 |
| ARMCM4_STM32L4/nvm.c | Bootloader non-volatile memory driver source file | 726 |
| ARMCM4_TM4C/nvm.c | Bootloader non-volatile memory driver source file | 729 |
| ARMCM4_XMC4/nvm.c | Bootloader non-volatile memory driver source file | 732 |
| ARMCM7_STM32F7/nvm.c | Bootloader non-volatile memory driver source file | 735 |
| ARMCM7_STM32H7/nvm.c | Bootloader non-volatile memory driver source file | 738 |

| | |
|--|-----|
| HCS12/nvm.c | |
| Bootloader non-volatile memory driver source file | 741 |
| TRICORE_TC2/nvm.c | |
| Bootloader non-volatile memory driver source file | 744 |
| TRICORE_TC3/nvm.c | |
| Bootloader non-volatile memory driver source file | 747 |
| _template/rs232.c | |
| Bootloader RS232 communication interface source file | 750 |
| ARMCM0_S32K11/rs232.c | |
| Bootloader RS232 communication interface source file | 751 |
| ARMCM0_STM32C0/rs232.c | |
| Bootloader RS232 communication interface source file | 752 |
| ARMCM0_STM32F0/rs232.c | |
| Bootloader RS232 communication interface source file | 752 |
| ARMCM0_STM32G0/rs232.c | |
| Bootloader RS232 communication interface source file | 753 |
| ARMCM0_STM32L0/rs232.c | |
| Bootloader RS232 communication interface source file | 753 |
| ARMCM0_XMC1/rs232.c | |
| Bootloader RS232 communication interface source file | 754 |
| ARMCM33_STM32H5/rs232.c | |
| Bootloader RS232 communication interface source file | 754 |
| ARMCM33_STM32L5/rs232.c | |
| Bootloader RS232 communication interface source file | 755 |
| ARMCM33_STM32U5/rs232.c | |
| Bootloader RS232 communication interface source file | 755 |
| ARMCM3_EFM32/rs232.c | |
| Bootloader RS232 communication interface source file | 756 |
| ARMCM3_LM3S/rs232.c | |
| Bootloader RS232 communication interface source file | 756 |
| ARMCM3_STM32F1/rs232.c | |
| Bootloader RS232 communication interface source file | 757 |
| ARMCM3_STM32F2/rs232.c | |
| Bootloader RS232 communication interface source file | 757 |
| ARMCM3_STM32L1/rs232.c | |
| Bootloader RS232 communication interface source file | 758 |
| ARMCM4_S32K14/rs232.c | |
| Bootloader RS232 communication interface source file | 758 |
| ARMCM4_STM32F3/rs232.c | |
| Bootloader RS232 communication interface source file | 759 |
| ARMCM4_STM32F4/rs232.c | |
| Bootloader RS232 communication interface source file | 759 |
| ARMCM4_STM32G4/rs232.c | |
| Bootloader RS232 communication interface source file | 760 |
| ARMCM4_STM32L4/rs232.c | |
| Bootloader RS232 communication interface source file | 760 |
| ARMCM4_TM4C/rs232.c | |
| Bootloader RS232 communication interface source file | 761 |
| ARMCM4_XMC4/rs232.c | |
| Bootloader RS232 communication interface source file | 761 |
| ARMCM7_STM32F7/rs232.c | |
| Bootloader RS232 communication interface source file | 762 |
| ARMCM7_STM32H7/rs232.c | |
| Bootloader RS232 communication interface source file | 762 |
| HCS12/rs232.c | |
| Bootloader RS232 communication interface source file | 763 |
| TRICORE_TC2/rs232.c | |
| Bootloader RS232 communication interface source file | 763 |

| | |
|--|-----|
| TRICORE_TC3/rs232.c | |
| Bootloader RS232 communication interface source file | 764 |
| _template/timer.c | |
| Bootloader timer driver source file | 764 |
| ARMCM0_S32K11/timer.c | |
| Bootloader timer driver source file | 767 |
| ARMCM0_STM32C0/timer.c | |
| Bootloader timer driver source file | 769 |
| ARMCM0_STM32F0/timer.c | |
| Bootloader timer driver source file | 772 |
| ARMCM0_STM32G0/timer.c | |
| Bootloader timer driver source file | 774 |
| ARMCM0_STM32L0/timer.c | |
| Bootloader timer driver source file | 777 |
| ARMCM0_XMC1/timer.c | |
| Bootloader timer driver source file | 780 |
| ARMCM33_STM32H5/timer.c | |
| Bootloader timer driver source file | 782 |
| ARMCM33_STM32L5/timer.c | |
| Bootloader timer driver source file | 785 |
| ARMCM33_STM32U5/timer.c | |
| Bootloader timer driver source file | 787 |
| ARMCM3_EFM32/timer.c | |
| Bootloader timer driver source file | 790 |
| ARMCM3_LM3S/timer.c | |
| Bootloader timer driver source file | 792 |
| ARMCM3_STM32F1/timer.c | |
| Bootloader timer driver source file | 794 |
| ARMCM3_STM32F2/timer.c | |
| Bootloader timer driver source file | 797 |
| ARMCM3_STM32L1/timer.c | |
| Bootloader timer driver source file | 799 |
| ARMCM4_S32K14/timer.c | |
| Bootloader timer driver source file | 802 |
| ARMCM4_STM32F3/timer.c | |
| Bootloader timer driver source file | 804 |
| ARMCM4_STM32F4/timer.c | |
| Bootloader timer driver source file | 807 |
| ARMCM4_STM32G4/timer.c | |
| Bootloader timer driver source file | 809 |
| ARMCM4_STM32L4/timer.c | |
| Bootloader timer driver source file | 812 |
| ARMCM4_TM4C/timer.c | |
| Bootloader timer driver source file | 815 |
| ARMCM4_XMC4/timer.c | |
| Bootloader timer driver source file | 817 |
| ARMCM7_STM32F7/timer.c | |
| Bootloader timer driver source file | 819 |
| ARMCM7_STM32H7/timer.c | |
| Bootloader timer driver source file | 822 |
| HCS12/timer.c | |
| Bootloader timer driver source file | 824 |
| TRICORE_TC2/timer.c | |
| Bootloader timer driver source file | 827 |
| TRICORE_TC3/timer.c | |
| Bootloader timer driver source file | 829 |
| _template/types.h | |
| Bootloader types header file | 831 |

| | | |
|---|--|-----|
| ARMCM0_S32K11/types.h | Bootloader types header file | 833 |
| ARMCM0_STM32C0/types.h | Bootloader types header file | 835 |
| ARMCM0_STM32F0/types.h | Bootloader types header file | 837 |
| ARMCM0_STM32G0/types.h | Bootloader types header file | 839 |
| ARMCM0_STM32L0/types.h | Bootloader types header file | 842 |
| ARMCM0_XMC1/types.h | Bootloader types header file | 844 |
| ARMCM33_STM32H5/types.h | Bootloader types header file | 846 |
| ARMCM33_STM32L5/types.h | Bootloader types header file | 848 |
| ARMCM33_STM32U5/types.h | Bootloader types header file | 851 |
| ARMCM3_EFM32/types.h | Bootloader types header file | 853 |
| ARMCM3_LM3S/types.h | Bootloader types header file | 855 |
| ARMCM3_STM32F1/types.h | Bootloader types header file | 857 |
| ARMCM3_STM32F2/types.h | Bootloader types header file | 859 |
| ARMCM3_STM32L1/types.h | Bootloader types header file | 861 |
| ARMCM4_S32K14/types.h | Bootloader types header file | 863 |
| ARMCM4_STM32F3/types.h | Bootloader types header file | 865 |
| ARMCM4_STM32F4/types.h | Bootloader types header file | 867 |
| ARMCM4_STM32G4/types.h | Bootloader types header file | 869 |
| ARMCM4_STM32L4/types.h | Bootloader types header file | 872 |
| ARMCM4_TM4C/types.h | Bootloader types header file | 874 |
| ARMCM4_XMC4/types.h | Bootloader types header file | 876 |
| ARMCM7_STM32F7/types.h | Bootloader types header file | 878 |
| ARMCM7_STM32H7/types.h | Bootloader types header file | 880 |
| HCS12/types.h | Bootloader types header file | 882 |
| TRICORE_TC2/types.h | Bootloader types header file | 884 |
| TRICORE_TC3/types.h | Bootloader types header file | 886 |
| _template/usb.c | Bootloader USB communication interface source file | 888 |
| ARMCM33_STM32H5/usb.c | Bootloader USB communication interface source file | 895 |
| ARMCM33_STM32L5/usb.c | Bootloader USB communication interface source file | 898 |

| | | |
|---------------------------------------|---|------|
| ARMCM33_STM32U5/usb.c | Bootloader USB communication interface source file | 904 |
| ARMCM3_STM32F1/usb.c | Bootloader USB communication interface source file | 912 |
| ARMCM3_STM32F2/usb.c | Bootloader USB communication interface source file | 918 |
| ARMCM4_STM32F3/usb.c | Bootloader USB communication interface source file | 924 |
| ARMCM4_STM32F4/usb.c | Bootloader USB communication interface source file | 932 |
| ARMCM4_STM32L4/usb.c | Bootloader USB communication interface source file | 938 |
| ARMCM4_TM4C/usb.c | Bootloader USB communication interface source file | 944 |
| ARMCM4_XMC4/usb.c | Bootloader USB communication interface source file | 947 |
| ARMCM7_STM32F7/usb.c | Bootloader USB communication interface source file | 950 |
| ARMCM7_STM32H7/usb.c | Bootloader USB communication interface source file | 958 |
| asserts.c | Bootloader assertion module source file | 964 |
| asserts.h | Bootloader assertion module header file | 965 |
| backdoor.c | Bootloader backdoor entry source file | 966 |
| backdoor.h | Bootloader backdoor entry header file | 967 |
| boot.c | Bootloader core module source file | 969 |
| boot.h | Bootloader core module header file | 970 |
| can.h | Bootloader CAN communication interface header file | 971 |
| com.c | Bootloader communication interface source file | 973 |
| com.h | Bootloader communication interface header file | 976 |
| cop.c | Bootloader watchdog module source file | 981 |
| cop.h | Bootloader watchdog module header file | 982 |
| cpu.h | Bootloader cpu module header file | 983 |
| file.c | Bootloader file system interface source file | 986 |
| file.h | Bootloader file system interface header file | 992 |
| flash_ecc.c | Bootloader flash driver source file for HCS12 derivatives with error correction code in flash memory, such as the HCS12Pxx. This flash memory uses a different addressing scheme than other HCS12 derivatives | 997 |
| mb.c | Bootloader Modbus communication interface source file | 1008 |
| mb.h | Bootloader Modbus communication interface header file | 1008 |
| net.c | Bootloader TCP/IP network communication interface source file | 1009 |

| | | |
|--|---|------|
| net.h | Bootloader TCP/IP network communication interface header file | 1012 |
| nvm.h | Bootloader non-volatile memory driver header file | 1014 |
| plausibility.h | Bootloader plausibility check header file, for checking the configuration at compile time | 1017 |
| rs232.h | Bootloader RS232 communication interface header file | 1017 |
| timer.h | Bootloader timer driver header file | 1017 |
| TRICORE_TC2/Tasking/ram_func.h | RAM function macros header file | 1020 |
| TRICORE_TC3/Tasking/ram_func.h | RAM function macros header file | 1021 |
| usb.h | Bootloader USB communication interface header file | 1021 |
| xcp.c | XCP 1.0 protocol core source file | 1023 |
| xcp.h | XCP 1.0 protocol core header file | 1036 |

Chapter 5

Module Documentation

5.1 CAN driver of a port

This module implements the CAN driver of a microcontroller port.

Files

- file [_template/can.c](#)
Bootloader CAN communication interface source file.

5.1.1 Detailed Description

This module implements the CAN driver of a microcontroller port.

For the most parts, this driver is already implemented. The only parts that need porting are the CAN initialization, CAN message reception and CAN message transmission.

5.2 CPU driver of a port

This module implements the CPU driver of a microcontroller port.

Files

- file [_template/cpu.c](#)
Bootloader cpu module source file.

5.2.1 Detailed Description

This module implements the CPU driver of a microcontroller port.

5.3 Flash driver of a port

This module implements the flash EEPROM memory driver of a microcontroller port.

Files

- file [_template/flash.c](#)
Bootloader flash driver source file.
- file [_template/flash.h](#)
Bootloader flash driver header file.

5.3.1 Detailed Description

This module implements the flash EEPROM memory driver of a microcontroller port.

The flash driver manages the actual erase and program operations on the flash EEPROM and the signature checksum. The signature checksum is a 32-bit value in the user program. It is used as a flag to determine if a user program is present or not. Newly programmed data is always first buffered in RAM buffers with a size of `FLASH_WRITE_BLOCK_SIZE`. This driver manages a second RAM buffer of the same size, called the `bootBlock`. The `bootBlock` buffers program data that includes the interrupt vector table and the 32-bit signature checksum. The signature checksum value is written as the last step during a firmware update, hence the need for the `bootBlock`. Note that the majority of this flash driver can be used as is. The only parts that need to be updated / implemented are:

- Macros `FLASH_WRITE_BLOCK_SIZE` and `BOOT_FLASH_VECTOR_TABLE_CS_OFFSET`.
- The `flashLayout[]`-array contents.
- The functions [FlashEraseSectors\(\)](#) and [FlashWriteBlock\(\)](#).
- The functions [FlashWriteChecksum\(\)](#) and [FlashVerifyChecksum\(\)](#).

5.4 Compiler specifics of a port

This module implements the compiler specific parts of a microcontroller port.

Files

- file [_template/GCC/cpu_comp.c](#)
Bootloader cpu module source file.

5.4.1 Detailed Description

This module implements the compiler specific parts of a microcontroller port.

5.5 Modbus RTU driver of a port

This module implements the Modbus RTU driver of a microcontroller port.

Files

- file [_template/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.

5.5.1 Detailed Description

This module implements the Modbus RTU driver of a microcontroller port.

For the most parts, this driver is already implemented. The only parts that need porting are the UART initialization, byte reception and byte transmission.

5.6 Non-volatile memory driver of a port

This module implements the non-volatile memory driver of a microcontroller port. Note that the default implementation is for a microcontroller that has internal flash memory. At the time of this writing pretty much all microcontrollers use flash EEPROM as non-volatile memory to store the program code. Assuming that this is also the case for the microcontroller for which the port is developed, nothing needs to be modified in this source file.

Files

- file [_template/nvm.c](#)
Bootloader non-volatile memory driver source file.

5.6.1 Detailed Description

This module implements the non-volatile memory driver of a microcontroller port. Note that the default implementation is for a microcontroller that has internal flash memory. At the time of this writing pretty much all microcontrollers use flash EEPROM as non-volatile memory to store the program code. Assuming that this is also the case for the microcontroller for which the port is developed, nothing needs to be modified in this source file.

5.7 RS232 UART driver of a port

This module implements the RS232 UART driver of a microcontroller port.

Files

- file [_template/rs232.c](#)
Bootloader RS232 communication interface source file.

5.7.1 Detailed Description

This module implements the RS232 UART driver of a microcontroller port.

For the most parts, this driver is already implemented. The only parts that need porting are the UART initialization, byte reception and byte transmission.

5.8 Target Port Template

Target dependent code as a template for new microcontroller ports.

Modules

- [CAN driver of a port](#)
This module implements the CAN driver of a microcontroller port.
- [CPU driver of a port](#)
This module implements the CPU driver of a microcontroller port.
- [Flash driver of a port](#)
This module implements the flash EEPROM memory driver of a microcontroller port.
- [Compiler specifics of a port](#)
This module implements the compiler specific parts of a microcontroller port.
- [Modbus RTU driver of a port](#)
This module implements the Modbus RTU driver of a microcontroller port.
- [Non-volatile memory driver of a port](#)
This module implements the non-volatile memory driver of a microcontroller port. Note that the default implementation if for a microcontroller that has internal flash memory. At the time of this writing pretty much all microcontrollers use flash EEPROM as non-volatile memory to store the program code. Assuming that this is also the case for the microcontroller for which the port is developed, nothing needs to be modified in this source file.
- [RS232 UART driver of a port](#)
This module implements the RS232 UART driver of a microcontroller port.
- [Timer driver of a port](#)
This module implements the timer memory driver of a microcontroller port.
- [Type definitions of a port](#)
This module implements the variable type definitions of a microcontroller port.
- [USB driver of a port](#)
This module implements the USB driver of a microcontroller port.

5.8.1 Detailed Description

Target dependent code as a template for new microcontroller ports.

This module serves as a template to implement the bootloader's target dependent part for a specific microcontroller family. It can be copied and used as a foundation when developing new microcontroller ports. Note that the parts of a port that need to be implemented are described as a source code comment that starts with "TODO ##Port".

5.9 Timer driver of a port

This module implements the timer memory driver of a microcontroller port.

Files

- file [_template/timer.c](#)
Bootloader timer driver source file.

5.9.1 Detailed Description

This module implements the timer memory driver of a microcontroller port.

The timer driver implements a polling based 1 millisecond timer. It provides the time base for all timing related parts of the bootloader. The bootloader calls the function [TimerUpdate\(\)](#) continuously to check if the next millisecond period passed.

5.10 Type definitions of a port

This module implements the variable type definitions of a microcontroller port.

Files

- file [_template/types.h](#)
Bootloader types header file.

5.10.1 Detailed Description

This module implements the variable type definitions of a microcontroller port.

5.11 USB driver of a port

This module implements the USB driver of a microcontroller port.

Files

- file [_template/usb.c](#)
Bootloader USB communication interface source file.

5.11.1 Detailed Description

This module implements the USB driver of a microcontroller port.

The USB driver makes user of bulk communications only, by means of two endpoints. This driver already implements FIFO buffers for the endpoint data including utility functions for managing these FIFOs. Note that the USB descriptor configuration and other enumeration related configuration is typically stored with the bootloader demo application and is not implemented in this driver directly. Basically, whenever the USB communication stack is available to transmit new data on the IN-endpoint, function [UsbTransmitPipeBulkIN\(\)](#) can be called, which checks if there is data to transmit in the FIFO buffer and then needs to copy it to the IN-endpoint buffer. Whenever the USB communication stack signals that new data was received on the OUT-endpoint, [UsbReceivePipeBulkOUT\(\)](#) can be called which then copies the data from the OUT-endpoint buffer into the FIFO buffer.

5.12 Target ARMCM0 S32K11

Target dependent code for the NXP ARMCM0 S32K11x microcontroller family.

Files

- file [ARMCM0_S32K11/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM0_S32K11/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM0_S32K11/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM0_S32K11/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM0_S32K11/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_S32K11/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_S32K11/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM0_S32K11/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM0_S32K11/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM0_S32K11/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM0_S32K11/types.h](#)
Bootloader types header file.

5.12.1 Detailed Description

Target dependent code for the NXP ARMCM0 S32K11x microcontroller family.

This module implements the bootloader's target dependent part for the NXP ARMCM0 S32K11x microcontroller family.

5.13 Target ARMCM0 STM32C0

Target dependent code for the ARMCM0 STM32C0 microcontroller family.

Files

- file [ARMCM0_STM32C0/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32C0/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM0_STM32C0/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM0_STM32C0/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32C0/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32C0/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32C0/modbus.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM0_STM32C0/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM0_STM32C0/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM0_STM32C0/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM0_STM32C0/types.h](#)
Bootloader types header file.

5.13.1 Detailed Description

Target dependent code for the ARMCM0 STM32C0 microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM0 STM32C0 microcontroller family.

5.14 Target ARMCM0 STM32F0

Target dependent code for the ARMCM0 STM32F0 microcontroller family.

Files

- file [ARMCM0_STM32F0/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM0_STM32F0/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32F0/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM0_STM32F0/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM0_STM32F0/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32F0/IAR/cpu_comp.c](#)

- file [ARMCM0_STM32F0/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32F0/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM0_STM32F0/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM0_STM32F0/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM0_STM32F0/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM0_STM32F0/types.h](#)
Bootloader types header file.

5.14.1 Detailed Description

Target dependent code for the ARMCM0 STM32F0 microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM0 STM32F0 microcontroller family.

5.15 Target ARMCM0 STM32G0

Target dependent code for the ARMCM0 STM32G0 microcontroller family.

Files

- file [ARMCM0_STM32G0/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM0_STM32G0/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32G0/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM0_STM32G0/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM0_STM32G0/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32G0/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32G0/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32G0/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM0_STM32G0/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM0_STM32G0/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM0_STM32G0/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM0_STM32G0/types.h](#)
Bootloader types header file.

5.15.1 Detailed Description

Target dependent code for the ARMCM0 STM32G0 microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM0 STM32G0 microcontroller family.

5.16 Target ARMCM0 STM32L0

Target dependent code for the ARMCM0 STM32L0 microcontroller family.

Files

- file [ARMCM0_STM32L0/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32L0/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM0_STM32L0/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM0_STM32L0/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32L0/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32L0/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_STM32L0/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM0_STM32L0/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM0_STM32L0/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM0_STM32L0/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM0_STM32L0/types.h](#)
Bootloader types header file.

5.16.1 Detailed Description

Target dependent code for the ARMCM0 STM32L0 microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM0 STM32L0 microcontroller family.

5.17 Target ARMCM0 XMC1

Target dependent code for the ARMCM0 XMC1xxx microcontroller family.

Files

- file [ARMCM0_XMC1/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM0_XMC1/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM0_XMC1/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM0_XMC1/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM0_XMC1/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_XMC1/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_XMC1/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM0_XMC1/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM0_XMC1/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM0_XMC1/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM0_XMC1/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM0_XMC1/types.h](#)
Bootloader types header file.

5.17.1 Detailed Description

Target dependent code for the ARMCM0 XMC1xxx microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM0 XMC1xxx microcontroller family.

5.18 Target ARMCM33 STM32H5

Target dependent code for the ARMCM33 STM32H5 microcontroller family.

Files

- file [ARMCM33_STM32H5/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM33_STM32H5/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM33_STM32H5/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM33_STM32H5/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM33_STM32H5/GCC/cpu_comp.c](#)

- file [ARCMC33_STM32H5/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARCMC33_STM32H5/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARCMC33_STM32H5/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARCMC33_STM32H5/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARCMC33_STM32H5/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARCMC33_STM32H5/timer.c](#)
Bootloader timer driver source file.
- file [ARCMC33_STM32H5/types.h](#)
Bootloader types header file.
- file [ARCMC33_STM32H5/usb.c](#)
Bootloader USB communication interface source file.

5.18.1 Detailed Description

Target dependent code for the ARCMC33 STM32H5 microcontroller family.

This module implements the bootloader's target dependent part for the ARCMC33 STM32H5 microcontroller family.

5.19 Target ARCMC33 STM32L5

Target dependent code for the ARCMC33 STM32L5 microcontroller family.

Files

- file [ARCMC33_STM32L5/can.c](#)
Bootloader CAN communication interface source file.
- file [ARCMC33_STM32L5/cpu.c](#)
Bootloader cpu module source file.
- file [ARCMC33_STM32L5/flash.c](#)
Bootloader flash driver source file.
- file [ARCMC33_STM32L5/flash.h](#)
Bootloader flash driver header file.
- file [ARCMC33_STM32L5/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARCMC33_STM32L5/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARCMC33_STM32L5/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARCMC33_STM32L5/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.

- file [ARMCM33_STM32L5/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM33_STM32L5/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM33_STM32L5/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM33_STM32L5/types.h](#)
Bootloader types header file.
- file [ARMCM33_STM32L5/usb.c](#)
Bootloader USB communication interface source file.

5.19.1 Detailed Description

Target dependent code for the ARMCM33 STM32L5 microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM33 STM32L5 microcontroller family.

5.20 Target ARMCM33 STM32U5

Target dependent code for the ARMCM33 STM32U5 microcontroller family.

Files

- file [ARMCM33_STM32U5/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM33_STM32U5/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM33_STM32U5/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM33_STM32U5/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM33_STM32U5/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM33_STM32U5/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM33_STM32U5/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM33_STM32U5/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM33_STM32U5/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM33_STM32U5/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM33_STM32U5/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM33_STM32U5/types.h](#)
Bootloader types header file.
- file [ARMCM33_STM32U5/usb.c](#)
Bootloader USB communication interface source file.

5.20.1 Detailed Description

Target dependent code for the ARCMCM33 STM32U5 microcontroller family.

This module implements the bootloader's target dependent part for the ARCMCM33 STM32U5 microcontroller family.

5.21 Target ARCMCM3 EFM32

Target dependent code for the ARCMCM3 EFM32 microcontroller family.

Files

- file [ARCMCM3_EFM32/cpu.c](#)
Bootloader cpu module source file.
- file [ARCMCM3_EFM32/flash.c](#)
Bootloader flash driver source file.
- file [ARCMCM3_EFM32/flash.h](#)
Bootloader flash driver header file.
- file [ARCMCM3_EFM32/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARCMCM3_EFM32/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARCMCM3_EFM32/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARCMCM3_EFM32/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARCMCM3_EFM32/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARCMCM3_EFM32/timer.c](#)
Bootloader timer driver source file.
- file [ARCMCM3_EFM32/types.h](#)
Bootloader types header file.

5.21.1 Detailed Description

Target dependent code for the ARCMCM3 EFM32 microcontroller family.

This module implements the bootloader's target dependent part for the ARCMCM3 EFM32 microcontroller family.

5.22 Target ARCMCM3 LM3S

Target dependent code for the ARCMCM3 LM3S microcontroller family.

Files

- file [ARMCM3_LM3S/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM3_LM3S/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM3_LM3S/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM3_LM3S/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM3_LM3S/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM3_LM3S/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM3_LM3S/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM3_LM3S/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM3_LM3S/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM3_LM3S/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM3_LM3S/types.h](#)
Bootloader types header file.

5.22.1 Detailed Description

Target dependent code for the ARMCM3 LM3S microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM3 LM3S microcontroller family.

5.23 Target ARMCM3 STM32F1

Target dependent code for the ARMCM3 STM32F1 microcontroller family.

Files

- file [ARMCM3_STM32F1/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM3_STM32F1/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM3_STM32F1/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM3_STM32F1/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM3_STM32F1/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM3_STM32F1/IAR/cpu_comp.c](#)

- file [ARMCM3_STM32F1/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM3_STM32F1/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM3_STM32F1/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM3_STM32F1/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM3_STM32F1/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM3_STM32F1/types.h](#)
Bootloader types header file.
- file [ARMCM3_STM32F1/usb.c](#)
Bootloader USB communication interface source file.

5.23.1 Detailed Description

Target dependent code for the ARMCM3 STM32F1 microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM3 STM32F1 microcontroller family.

5.24 Target ARMCM3 STM32F2

Target dependent code for the ARMCM3 STM32F2xx microcontroller family.

Files

- file [ARMCM3_STM32F2/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM3_STM32F2/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM3_STM32F2/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM3_STM32F2/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM3_STM32F2/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM3_STM32F2/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM3_STM32F2/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM3_STM32F2/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM3_STM32F2/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM3_STM32F2/rs232.c](#)

- file [ARMCM3_STM32F2/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM3_STM32F2/types.h](#)
Bootloader types header file.
- file [ARMCM3_STM32F2/usb.c](#)
Bootloader USB communication interface source file.

5.24.1 Detailed Description

Target dependent code for the ARMCM3 STM32F2xx microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM3 STM32F2xx microcontroller family.

5.25 Target ARMCM3 STM32L1

Target dependent code for the ARMCM3 STM32L1 microcontroller family.

Files

- file [ARMCM3_STM32L1/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM3_STM32L1/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM3_STM32L1/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM3_STM32L1/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM3_STM32L1/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM3_STM32L1/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM3_STM32L1/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM3_STM32L1/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM3_STM32L1/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM3_STM32L1/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM3_STM32L1/types.h](#)
Bootloader types header file.

5.25.1 Detailed Description

Target dependent code for the ARMCM3 STM32L1 microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM3 STM32L1 microcontroller family.

5.26 Target ARMCM4 S32K14

Target dependent code for the NXP ARMCM4 S32K14x microcontroller family.

Files

- file [ARMCM4_S32K14/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM4_S32K14/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM4_S32K14/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM4_S32K14/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM4_S32K14/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_S32K14/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_S32K14/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM4_S32K14/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM4_S32K14/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM4_S32K14/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM4_S32K14/types.h](#)
Bootloader types header file.

5.26.1 Detailed Description

Target dependent code for the NXP ARMCM4 S32K14x microcontroller family.

This module implements the bootloader's target dependent part for the NXP ARMCM4 S32K14x microcontroller family.

5.27 Target ARMCM4 STM32F3

Target dependent code for the ARMCM4 STM32F3 microcontroller family.

Files

- file [ARMCM4_STM32F3/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM4_STM32F3/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM4_STM32F3/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM4_STM32F3/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM4_STM32F3/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_STM32F3/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_STM32F3/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_STM32F3/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM4_STM32F3/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM4_STM32F3/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM4_STM32F3/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM4_STM32F3/types.h](#)
Bootloader types header file.
- file [ARMCM4_STM32F3/usb.c](#)
Bootloader USB communication interface source file.

5.27.1 Detailed Description

Target dependent code for the ARMCM4 STM32F3 microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM4 STM32F3 microcontroller family.

5.28 Target ARMCM4 STM32F4

Target dependent code for the ARMCM4 STM32F4 microcontroller family.

Files

- file [ARMCM4_STM32F4/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM4_STM32F4/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM4_STM32F4/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM4_STM32F4/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM4_STM32F4/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_STM32F4/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_STM32F4/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_STM32F4/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM4_STM32F4/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM4_STM32F4/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM4_STM32F4/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM4_STM32F4/types.h](#)
Bootloader types header file.
- file [ARMCM4_STM32F4/usb.c](#)
Bootloader USB communication interface source file.

5.28.1 Detailed Description

Target dependent code for the ARMCM4 STM32F4 microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM4 STM32F4 microcontroller family.

5.29 Target ARMCM4 STM32G4

Target dependent code for the ARMCM4 STM32G4 microcontroller family.

Files

- file [ARMCM4_STM32G4/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM4_STM32G4/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM4_STM32G4/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM4_STM32G4/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM4_STM32G4/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_STM32G4/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_STM32G4/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_STM32G4/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM4_STM32G4/nvram.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM4_STM32G4/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM4_STM32G4/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM4_STM32G4/types.h](#)
Bootloader types header file.

5.29.1 Detailed Description

Target dependent code for the ARMCM4 STM32G4 microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM4 STM32G4 microcontroller family.

5.30 Target ARMCM4 STM32L4

Target dependent code for the ARMCM4 STM32L4 microcontroller family.

Files

- file [ARMCM4_STM32L4/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM4_STM32L4/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM4_STM32L4/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM4_STM32L4/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM4_STM32L4/GCC/cpu_comp.c](#)

- file [ARCMCM4_STM32L4/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARCMCM4_STM32L4/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARCMCM4_STM32L4/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARCMCM4_STM32L4/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARCMCM4_STM32L4/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARCMCM4_STM32L4/timer.c](#)
Bootloader timer driver source file.
- file [ARCMCM4_STM32L4/types.h](#)
Bootloader types header file.
- file [ARCMCM4_STM32L4/usb.c](#)
Bootloader USB communication interface source file.

5.30.1 Detailed Description

Target dependent code for the ARCMCM4 STM32L4 microcontroller family.

This module implements the bootloader's target dependent part for the ARCMCM4 STM32L4 microcontroller family.

5.31 Target ARCMCM4 TM4C

Target dependent code for the ARCMCM4 TM4C microcontroller family.

Files

- file [ARCMCM4_TM4C/cpu.c](#)
Bootloader cpu module source file.
- file [ARCMCM4_TM4C/flash.c](#)
Bootloader flash driver source file.
- file [ARCMCM4_TM4C/flash.h](#)
Bootloader flash driver header file.
- file [ARCMCM4_TM4C/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARCMCM4_TM4C/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARCMCM4_TM4C/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARCMCM4_TM4C/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARCMCM4_TM4C/timer.c](#)
Bootloader timer driver source file.
- file [ARCMCM4_TM4C/types.h](#)
Bootloader types header file.
- file [ARCMCM4_TM4C/usb.c](#)
Bootloader USB communication interface source file.

5.31.1 Detailed Description

Target dependent code for the ARMCM4 TM4C microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM4 TM4C microcontroller family.

5.32 Target ARMCM4 XMC4

Target dependent code for the ARMCM4 XMC4xxx microcontroller family.

Files

- file [ARMCM4_XMC4/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM4_XMC4/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM4_XMC4/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM4_XMC4/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM4_XMC4/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_XMC4/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_XMC4/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM4_XMC4/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM4_XMC4/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM4_XMC4/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM4_XMC4/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM4_XMC4/types.h](#)
Bootloader types header file.
- file [ARMCM4_XMC4/usb.c](#)
Bootloader USB communication interface source file.

5.32.1 Detailed Description

Target dependent code for the ARMCM4 XMC4xxx microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM4 XMC4xxx microcontroller family.

5.33 Target ARMCM7 STM32F7

Target dependent code for the ARMCM7 STM32F7 microcontroller family.

Files

- file [ARMCM7_STM32F7/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM7_STM32F7/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM7_STM32F7/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM7_STM32F7/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM7_STM32F7/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM7_STM32F7/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM7_STM32F7/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM7_STM32F7/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM7_STM32F7/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM7_STM32F7/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM7_STM32F7/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM7_STM32F7/types.h](#)
Bootloader types header file.
- file [ARMCM7_STM32F7/usb.c](#)
Bootloader USB communication interface source file.

5.33.1 Detailed Description

Target dependent code for the ARMCM7 STM32F7 microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM7 STM32F7 microcontroller family.

5.34 Target ARMCM7 STM32H7

Target dependent code for the ARMCM7 STM32H7 microcontroller family.

Files

- file [ARMCM7_STM32H7/can.c](#)
Bootloader CAN communication interface source file.
- file [ARMCM7_STM32H7/cpu.c](#)
Bootloader cpu module source file.
- file [ARMCM7_STM32H7/flash.c](#)
Bootloader flash driver source file.
- file [ARMCM7_STM32H7/flash.h](#)
Bootloader flash driver header file.
- file [ARMCM7_STM32H7/GCC/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM7_STM32H7/IAR/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM7_STM32H7/Keil/cpu_comp.c](#)
Bootloader cpu module source file.
- file [ARMCM7_STM32H7/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [ARMCM7_STM32H7/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [ARMCM7_STM32H7/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [ARMCM7_STM32H7/timer.c](#)
Bootloader timer driver source file.
- file [ARMCM7_STM32H7/types.h](#)
Bootloader types header file.
- file [ARMCM7_STM32H7/usb.c](#)
Bootloader USB communication interface source file.

5.34.1 Detailed Description

Target dependent code for the ARMCM7 STM32H7 microcontroller family.

This module implements the bootloader's target dependent part for the ARMCM7 STM32H7 microcontroller family.

5.35 Bootloader Core

Target independent code.

Files

- file **asserts.c**
 Bootloader assertion module source file.
- file **asserts.h**
 Bootloader assertion module header file.
- file **backdoor.c**
 Bootloader backdoor entry source file.
- file **backdoor.h**
 Bootloader backdoor entry header file.
- file **boot.c**
 Bootloader core module source file.
- file **boot.h**
 Bootloader core module header file.
- file **can.h**
 Bootloader CAN communication interface header file.
- file **com.c**
 Bootloader communication interface source file.
- file **com.h**
 Bootloader communication interface header file.
- file **cop.c**
 Bootloader watchdog module source file.
- file **cop.h**
 Bootloader watchdog module header file.
- file **cpu.h**
 Bootloader cpu module header file.
- file **file.c**
 Bootloader file system interface source file.
- file **file.h**
 Bootloader file system interface header file.
- file **mb.c**
 Bootloader Modbus communication interface source file.
- file **mb.h**
 Bootloader Modbus communication interface header file.
- file **net.c**
 Bootloader TCP/IP network communication interface source file.
- file **net.h**
 Bootloader TCP/IP network communication interface header file.
- file **nvm.h**
 Bootloader non-volatile memory driver header file.
- file **plausibility.h**
 Bootloader plausibility check header file, for checking the configuration at compile time.
- file **rs232.h**
 Bootloader RS232 communication interface header file.
- file **timer.h**
 Bootloader timer driver header file.
- file **usb.h**
 Bootloader USB communication interface header file.
- file **xcp.c**
 XCP 1.0 protocol core source file.
- file **xcp.h**
 XCP 1.0 protocol core header file.

5.35.1 Detailed Description

Target independent code.

The bootloader core contains the main functionality of the bootloader, independent of the microcontroller/compiler and independent of your specific application. There is generally no need for you to make changes here, unless you are adding new functionality such as the support for a new communication interface or a different communication protocol.

By default the XCP version 1.0 is used as the communication transport layer for remote firmware updates, for example via UART or CAN. Its official name is ASAM MCD-1 XCP V1.0.0 and it is a universal measurement and calibration protocol that defines a bus-independent, master-slave communication protocol to connect ECU's with calibration systems. More information can be found at <http://www.asam.net/>.

For local firmware updates, for example from a locally attached SD-card, the FATFS file system is used as a target independent interface to access files. More information can be found at http://elm-chan.org/fsw/ff/00index_e.html

5.36 Target HCS12

Target dependent code for the Freescale HCS12 microcontroller family.

Files

- file [HCS12/can.c](#)
Bootloader CAN communication interface source file.
- file [HCS12/CodeWarrior/cpu_comp.c](#)
Bootloader cpu module source file.
- file [HCS12/cpu.c](#)
Bootloader cpu module source file.
- file [HCS12/flash.c](#)
Bootloader flash driver source file.
- file [HCS12/flash.h](#)
Bootloader flash driver header file.
- file [flash_ecc.c](#)
Bootloader flash driver source file for HCS12 derivatives with error correction code in flash memory, such as the HCS12Pxx. This flash memory uses a different addressing scheme than other HCS12 derivatives.
- file [HCS12/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [HCS12/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [HCS12/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [HCS12/timer.c](#)
Bootloader timer driver source file.
- file [HCS12/types.h](#)
Bootloader types header file.

5.36.1 Detailed Description

Target dependent code for the Freescale HCS12 microcontroller family.

This module implements the bootloader's target dependent part for the Freescale HCS12 microcontroller family.

5.37 Bootloader Ports

Target dependent code.

Modules

- [Target Port Template](#)
Target dependent code as a template for new microcontroller ports.
- [Target ARMCM0 S32K11](#)
Target dependent code for the NXP ARMCM0 S32K11x microcontroller family.
- [Target ARMCM0 STM32C0](#)
Target dependent code for the ARMCM0 STM32C0 microcontroller family.
- [Target ARMCM0 STM32F0](#)
Target dependent code for the ARMCM0 STM32F0 microcontroller family.
- [Target ARMCM0 STM32G0](#)
Target dependent code for the ARMCM0 STM32G0 microcontroller family.
- [Target ARMCM0 STM32L0](#)
Target dependent code for the ARMCM0 STM32L0 microcontroller family.
- [Target ARMCM0 XMC1](#)
Target dependent code for the ARMCM0 XMC1xxx microcontroller family.
- [Target ARMCM33 STM32H5](#)
Target dependent code for the ARMCM33 STM32H5 microcontroller family.
- [Target ARMCM33 STM32L5](#)
Target dependent code for the ARMCM33 STM32L5 microcontroller family.
- [Target ARMCM33 STM32U5](#)
Target dependent code for the ARMCM33 STM32U5 microcontroller family.
- [Target ARMCM3 EFM32](#)
Target dependent code for the ARMCM3 EFM32 microcontroller family.
- [Target ARMCM3 LM3S](#)
Target dependent code for the ARMCM3 LM3S microcontroller family.
- [Target ARMCM3 STM32F1](#)
Target dependent code for the ARMCM3 STM32F1 microcontroller family.
- [Target ARMCM3 STM32F2](#)
Target dependent code for the ARMCM3 STM32F2xx microcontroller family.
- [Target ARMCM3 STM32L1](#)
Target dependent code for the ARMCM3 STM32L1 microcontroller family.
- [Target ARMCM4 S32K14](#)
Target dependent code for the NXP ARMCM4 S32K14x microcontroller family.
- [Target ARMCM4 STM32F3](#)
Target dependent code for the ARMCM4 STM32F3 microcontroller family.
- [Target ARMCM4 STM32F4](#)
Target dependent code for the ARMCM4 STM32F4 microcontroller family.

- [Target ARMCM4 STM32G4](#)
Target dependent code for the ARMCM4 STM32G4 microcontroller family.
- [Target ARMCM4 STM32L4](#)
Target dependent code for the ARMCM4 STM32L4 microcontroller family.
- [Target ARMCM4 TM4C](#)
Target dependent code for the ARMCM4 TM4C microcontroller family.
- [Target ARMCM4 XMC4](#)
Target dependent code for the ARMCM4 XMC4xxx microcontroller family.
- [Target ARMCM7 STM32F7](#)
Target dependent code for the ARMCM7 STM32F7 microcontroller family.
- [Target ARMCM7 STM32H7](#)
Target dependent code for the ARMCM7 STM32H7 microcontroller family.
- [Target HCS12](#)
Target dependent code for the Freescale HCS12 microcontroller family.
- [Target TRICORE TC2](#)
Target dependent code for the Infineon TriCore AURIX TC2 microcontroller family.
- [Target TRICORE TC3](#)
Target dependent code for the Infineon TriCore AURIX TC3 microcontroller family.

5.37.1 Detailed Description

Target dependent code.

The bootloader targets contain the microcontroller and compiler dependent parts of the bootloader. They are grouped per microcontroller family. This is the part that needs to be newly developed when porting the bootloader to a new microcontroller family.

5.38 Target TRICORE TC2

Target dependent code for the Infineon TriCore AURIX TC2 microcontroller family.

Files

- file [TRICORE_TC2/can.c](#)
Bootloader CAN communication interface source file.
- file [TRICORE_TC2/cpu.c](#)
Bootloader cpu module source file.
- file [TRICORE_TC2/flash.c](#)
Bootloader flash driver source file.
- file [TRICORE_TC2/flash.h](#)
Bootloader flash driver header file.
- file [TRICORE_TC2/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [TRICORE_TC2/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [TRICORE_TC2/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [TRICORE_TC2/Tasking/cpu_comp.c](#)

- file [TRICORE_TC2/Tasking/ram_func.h](#)
RAM function macros header file.
- file [TRICORE_TC2/timer.c](#)
Bootloader timer driver source file.
- file [TRICORE_TC2/types.h](#)
Bootloader types header file.

5.38.1 Detailed Description

Target dependent code for the Infineon TriCore AURIX TC2 microcontroller family.

This module implements the bootloader's target dependent part for the Infineon TriCore AURIX TC2 microcontroller family.

5.39 Target TRICORE TC3

Target dependent code for the Infineon TriCore AURIX TC3 microcontroller family.

Files

- file [TRICORE_TC3/can.c](#)
Bootloader CAN communication interface source file.
- file [TRICORE_TC3/cpu.c](#)
Bootloader cpu module source file.
- file [TRICORE_TC3/flash.c](#)
Bootloader flash driver source file.
- file [TRICORE_TC3/flash.h](#)
Bootloader flash driver header file.
- file [TRICORE_TC3/mbrtu.c](#)
Bootloader Modbus RTU communication interface source file.
- file [TRICORE_TC3/nvm.c](#)
Bootloader non-volatile memory driver source file.
- file [TRICORE_TC3/rs232.c](#)
Bootloader RS232 communication interface source file.
- file [TRICORE_TC3/Tasking/cpu_comp.c](#)
Bootloader cpu module source file.
- file [TRICORE_TC3/Tasking/ram_func.h](#)
RAM function macros header file.
- file [TRICORE_TC3/timer.c](#)
Bootloader timer driver source file.
- file [TRICORE_TC3/types.h](#)
Bootloader types header file.

5.39.1 Detailed Description

Target dependent code for the Infineon TriCore AURIX TC3 microcontroller family.

This module implements the bootloader's target dependent part for the Infineon TriCore AURIX TC3 microcontroller family.

Chapter 6

Data Structure Documentation

6.1 tCanBusTiming Struct Reference

Structure type for grouping CAN bus timing related information.

Data Fields

- `blt_int8u tseg1`
- `blt_int8u tseg2`
- `blt_int8u timeQuanta`
- `blt_int8u propSeg`
- `blt_int8u phaseSeg1`
- `blt_int8u phaseSeg2`

6.1.1 Detailed Description

Structure type for grouping CAN bus timing related information.

Structure type with the layout of the CAN bus timing registers.

6.1.2 Field Documentation

6.1.2.1 phaseSeg1

`blt_int8u phaseSeg1`

CAN phase segment 1

Referenced by [CanInit\(\)](#).

6.1.2.2 phaseSeg2

`blt_int8u` phaseSeg2

CAN phase segment 2

Referenced by [CanInit\(\)](#).

6.1.2.3 propSeg

`blt_int8u` propSeg

CAN propagation segment

Referenced by [CanInit\(\)](#).

6.1.2.4 timeQuanta

`blt_int8u` timeQuanta

Total number of time quanta

Referenced by [CanGetSpeedConfig\(\)](#).

6.1.2.5 tseg1

`blt_int8u` tseg1

CAN time segment 1

Referenced by [CanGetSpeedConfig\(\)](#).

6.1.2.6 tseg2

`blt_int8u tseg2`

CAN time segment 2

Referenced by [CanGetSpeedConfig\(\)](#).

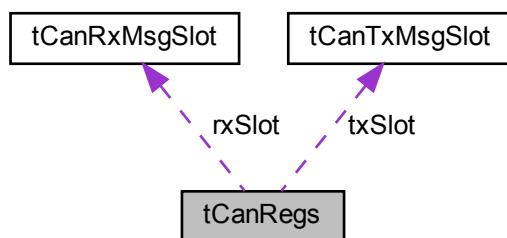
The documentation for this struct was generated from the following files:

- [_template/can.c](#)
- [ARMCM0_S32K11/can.c](#)
- [ARMCM0_STM32F0/can.c](#)
- [ARMCM0_STM32G0/can.c](#)
- [ARMCM33_STM32H5/can.c](#)
- [ARMCM33_STM32L5/can.c](#)
- [ARMCM33_STM32U5/can.c](#)
- [ARMCM3_STM32F1/can.c](#)
- [ARMCM3_STM32F2/can.c](#)
- [ARMCM4_S32K14/can.c](#)
- [ARMCM4_STM32F3/can.c](#)
- [ARMCM4_STM32F4/can.c](#)
- [ARMCM4_STM32G4/can.c](#)
- [ARMCM4_STM32L4/can.c](#)
- [ARMCM7_STM32F7/can.c](#)
- [ARMCM7_STM32H7/can.c](#)
- [HCS12/can.c](#)
- [TRICORE_TC2/can.c](#)
- [TRICORE_TC3/can.c](#)

6.2 tCanRegs Struct Reference

Structure type with the layout of the CAN related control registers.

Collaboration diagram for tCanRegs:



Data Fields

- volatile blt_int8u cctl0
- volatile blt_int8u cctl1
- volatile blt_int8u cbtr0
- volatile blt_int8u cbtr1
- volatile blt_int8u crflg
- volatile blt_int8u crier
- volatile blt_int8u cflg
- volatile blt_int8u ctier
- volatile blt_int8u ctarq
- volatile blt_int8u ctaak
- volatile blt_int8u ctbsel
- volatile blt_int8u cidac
- volatile blt_int8u dummy1 [2]
- volatile blt_int8u crxerr
- volatile blt_int8u ctxerr
- volatile blt_int8u cidar0
- volatile blt_int8u cidar1
- volatile blt_int8u cidar2
- volatile blt_int8u cidar3
- volatile blt_int8u cidmr0
- volatile blt_int8u cidmr1
- volatile blt_int8u cidmr2
- volatile blt_int8u cidmr3
- volatile blt_int8u cidar4
- volatile blt_int8u cidar5
- volatile blt_int8u cidar6
- volatile blt_int8u cidar7
- volatile blt_int8u cidmr4
- volatile blt_int8u cidmr5
- volatile blt_int8u cidmr6
- volatile blt_int8u cidmr7
- volatile tCanRxMsgSlot rxSlot
- volatile tCanTxMsgSlot txSlot

6.2.1 Detailed Description

Structure type with the layout of the CAN related control registers.

6.2.2 Field Documentation

6.2.2.1 cbtr0

```
volatile blt_int8u cbtr0
```

bus timing register 0

6.2.2.2 cbtr1

```
volatile blt_int8u cbtr1
```

bus timing register 1

6.2.2.3 cctl0

```
volatile blt_int8u cctl0
```

control register 0

6.2.2.4 cctl1

```
volatile blt_int8u cctl1
```

control register 1

6.2.2.5 cidac

```
volatile blt_int8u cidac
```

identifier acceptance control register

6.2.2.6 cidar0

```
volatile blt_int8u cidar0
```

identifier acceptance register 0

6.2.2.7 cidar1

```
volatile blt_int8u cidar1
```

identifier acceptance register 1

6.2.2.8 cidar2

```
volatile blt_int8u cidar2
```

identifier acceptance register 2

6.2.2.9 cedar3

volatile `blt_int8u` cedar3

identifier acceptance register 3

6.2.2.10 cedar4

volatile `blt_int8u` cedar4

identifier acceptance register 4

6.2.2.11 cedar5

volatile `blt_int8u` cedar5

identifier acceptance register 5

6.2.2.12 cedar6

volatile `blt_int8u` cedar6

identifier acceptance register 6

6.2.2.13 cedar7

volatile `blt_int8u` cedar7

identifier acceptance register 7

6.2.2.14 cidmr0

volatile `blt_int8u` cidmr0

identifier mask register 0

6.2.2.15 cidmr1

volatile `blt_int8u` cidmr1

identifier mask register 1

6.2.2.16 cidmr2

```
volatile blt_int8u cidmr2
```

identifier mask register 2

6.2.2.17 cidmr3

```
volatile blt_int8u cidmr3
```

identifier mask register 3

6.2.2.18 cidmr4

```
volatile blt_int8u cidmr4
```

identifier mask register 4

6.2.2.19 cidmr5

```
volatile blt_int8u cidmr5
```

identifier mask register 5

6.2.2.20 cidmr6

```
volatile blt_int8u cidmr6
```

identifier mask register 6

6.2.2.21 cidmr7

```
volatile blt_int8u cidmr7
```

identifier mask register 7

6.2.2.22 crflg

```
volatile blt_int8u crflg
```

receiver flag register

6.2.2.23 crier

volatile `blt_int8u` crier
receiver interrupt enable register

6.2.2.24 crxerr

volatile `blt_int8u` crxerr
receive error counter

6.2.2.25 ctaak

volatile `blt_int8u` ctaak
transmitter message abort control

6.2.2.26 ctarq

volatile `blt_int8u` ctarq
transmitter message abort control

6.2.2.27 ctbsel

volatile `blt_int8u` ctbsel
transmit buffer selection

6.2.2.28 ctflg

volatile `blt_int8u` ctflg
transmitter flag register

6.2.2.29 ctier

volatile `blt_int8u` ctier
transmitter interrupt enable register

6.2.2.30 ctxerr

volatile `blt_int8u` ctxerr

transmit error counter

6.2.2.31 dummy1

volatile `blt_int8u` dummy1[2]

reserved (2)

6.2.2.32 rxSlot

volatile `tCanRxMsgSlot` rxSlot

foreground receive message slot

6.2.2.33 txSlot

volatile `tCanTxMsgSlot` txSlot

foreground transmit message slot

The documentation for this struct was generated from the following file:

- [HCS12/can.c](#)

6.3 tCanRxMsgSlot Struct Reference

Structure type with the layout of a CAN reception message slot.

Data Fields

- volatile `blt_int8u` `idr` [4]
- volatile `blt_int8u` `dsr` [8]
- volatile `blt_int8u` `dlr`
- volatile `blt_int8u` `dummy`
- volatile `blt_int16u` `tstamp`

6.3.1 Detailed Description

Structure type with the layout of a CAN reception message slot.

6.3.2 Field Documentation

6.3.2.1 dlr

volatile `blt_int8u` dlr

data length register

6.3.2.2 dsr

volatile `blt_int8u` dsr[8]

data segment register 0..7

6.3.2.3 dummy

volatile `blt_int8u` dummy

unused

6.3.2.4 idr

volatile `blt_int8u` idr[4]

identifier register 0..3

6.3.2.5 tstamp

volatile `blt_int16u` tstamp

timestamp register

The documentation for this struct was generated from the following file:

- [HCS12/can.c](#)

6.4 tCanTxMsgSlot Struct Reference

Structure type with the layout of a CAN transmit message slot.

Data Fields

- volatile `blt_int8u` `idr` [4]
- volatile `blt_int8u` `dsr` [8]
- volatile `blt_int8u` `dlr`
- volatile `blt_int8u` `tbpr`
- volatile `blt_int16u` `tstamp`

6.4.1 Detailed Description

Structure type with the layout of a CAN transmit message slot.

6.4.2 Field Documentation

6.4.2.1 `dlr`

volatile `blt_int8u` `dlr`

data length register

6.4.2.2 `dsr`

volatile `blt_int8u` `dsr[8]`

data segment register 0..7

6.4.2.3 `idr`

volatile `blt_int8u` `idr[4]`

identifier register 0..3

6.4.2.4 `tbpr`

volatile `blt_int8u` `tbpr`

transmit buffer priority register

6.4.2.5 tstamp

volatile `blt_int16u` tstamp

timestamp register

The documentation for this struct was generated from the following file:

- [HCS12/can.c](#)

6.5 tFatFsObjects Struct Reference

Structure type for grouping FATFS related objects used by this module.

Data Fields

- FATFS `fs`
- FIL `file`

6.5.1 Detailed Description

Structure type for grouping FATFS related objects used by this module.

6.5.2 Field Documentation

6.5.2.1 file

FIL `file`

file object for firmware file

6.5.2.2 fs

FATFS `fs`

file system object for mounting

Referenced by [FileInit\(\)](#).

The documentation for this struct was generated from the following file:

- [file.c](#)

6.6 tFifoCtrl Struct Reference

Structure type for fifo control.

Data Fields

- `blt_int8u * startptr`
- `blt_int8u * endptr`
- `blt_int8u * readptr`
- `blt_int8u * writeptr`
- `blt_int8u length`
- `blt_int8u entries`
- `blt_int8u handle`
- `struct t_fifo_ctrl * fifoctrlptr`

6.6.1 Detailed Description

Structure type for fifo control.

6.6.2 Field Documentation

6.6.2.1 endptr

`blt_int8u * endptr`

pointer to end of buffer

Referenced by [UsbFifoMgrCreate\(\)](#).

6.6.2.2 entries

`blt_int8u entries`

of full buffer elements

Referenced by [UsbFifoMgrCreate\(\)](#), [UsbFifoMgrRead\(\)](#), [UsbFifoMgrScan\(\)](#), and [UsbFifoMgrWrite\(\)](#).

6.6.2.3 **fifoctrlptr**

`struct t_fifo_ctrl * fifoctrlptr`

pointer to free buffer control

Referenced by [UsbFifoMgrCreate\(\)](#), and [UsbFifoMgrInit\(\)](#).

6.6.2.4 **handle**

`blt_int8u handle`

handle of the buffer

Referenced by [UsbFifoMgrCreate\(\)](#), and [UsbFifoMgrInit\(\)](#).

6.6.2.5 **length**

`blt_int8u length`

number of buffer elements

Referenced by [UsbFifoMgrCreate\(\)](#).

6.6.2.6 **readptr**

`blt_int8u * readptr`

pointer to next read location

Referenced by [UsbFifoMgrCreate\(\)](#), and [UsbFifoMgrRead\(\)](#).

6.6.2.7 **startptr**

`blt_int8u * startptr`

pointer to start of buffer

Referenced by [UsbFifoMgrCreate\(\)](#), [UsbFifoMgrRead\(\)](#), and [UsbFifoMgrWrite\(\)](#).

6.6.2.8 writeptr

`blt_int8u * writeptr`

pointer to next free location

Referenced by [UsbFifoMgrCreate\(\)](#), and [UsbFifoMgrWrite\(\)](#).

The documentation for this struct was generated from the following files:

- [_template/usb.c](#)
- [ARCMC33_STM32L5/usb.c](#)
- [ARCMC33_STM32U5/usb.c](#)
- [ARCMC3_STM32F1/usb.c](#)
- [ARCMC3_STM32F2/usb.c](#)
- [ARCMC4_STM32F3/usb.c](#)
- [ARCMC4_STM32F4/usb.c](#)
- [ARCMC4_STM32L4/usb.c](#)
- [ARCMC7_STM32F7/usb.c](#)
- [ARCMC7_STM32H7/usb.c](#)

6.7 tFifoPipe Struct Reference

Structure type for a fifo pipe.

Data Fields

- `blt_int8u handle`
- `blt_int8u data [FIFO_PIPE_SIZE]`

6.7.1 Detailed Description

Structure type for a fifo pipe.

6.7.2 Field Documentation

6.7.2.1 data

`blt_int8u data`

fifo data buffer

Referenced by [UsbInit\(\)](#).

6.7.2.2 handle

`blt_int8u handle`

fifo handle

Referenced by [UsbInit\(\)](#), [UsbReceiveByte\(\)](#), [UsbReceivePipeBulkOUT\(\)](#), [UsbTransmitByte\(\)](#), and [UsbTransmitPipeBulkIN\(\)](#).

The documentation for this struct was generated from the following files:

- [_template/usb.c](#)
- [ARMCM33_STM32L5/usb.c](#)
- [ARMCM33_STM32U5/usb.c](#)
- [ARMCM3_STM32F1/usb.c](#)
- [ARMCM3_STM32F2/usb.c](#)
- [ARMCM4_STM32F3/usb.c](#)
- [ARMCM4_STM32F4/usb.c](#)
- [ARMCM4_STM32L4/usb.c](#)
- [ARMCM7_STM32F7/usb.c](#)
- [ARMCM7_STM32H7/usb.c](#)

6.8 tFileEraseInfo Struct Reference

Structure type with information for the memory erase opeartion.

Data Fields

- `blt_addr start_address`
- `blt_int32u total_size`

6.8.1 Detailed Description

Structure type with information for the memory erase opeartion.

6.8.2 Field Documentation

6.8.2.1 start_address

`blt_addr start_address`

erase start address

6.8.2.2 total_size

`blt_int32u total_size`

total number of bytes to erase

The documentation for this struct was generated from the following file:

- [file.c](#)

6.9 tFlashBlockInfo Struct Reference

Structure type for grouping flash block information.

6.9.1 Detailed Description

Structure type for grouping flash block information.

Programming is done per block of max FLASH_WRITE_BLOCK_SIZE. for this a flash block manager is implemented in this driver. this flash block manager depends on this flash block info structure. It holds the base address of the flash block and the data that should be programmed into the flash block. The .base_addr must be a multiple of FLASH_WRITE_BLOCK_SIZE.

The documentation for this struct was generated from the following files:

- [_template/flash.c](#)
- [ARMCM0_S32K11/flash.c](#)
- [ARMCM0_STM32C0/flash.c](#)
- [ARMCM0_STM32F0/flash.c](#)
- [ARMCM0_STM32G0/flash.c](#)
- [ARMCM0_STM32L0/flash.c](#)
- [ARMCM0_XMC1/flash.c](#)
- [ARMCM33_STM32H5/flash.c](#)
- [ARMCM33_STM32L5/flash.c](#)
- [ARMCM33_STM32U5/flash.c](#)
- [ARMCM3_EFM32/flash.c](#)
- [ARMCM3_LM3S/flash.c](#)
- [ARMCM3_STM32F1/flash.c](#)
- [ARMCM3_STM32F2/flash.c](#)
- [ARMCM3_STM32L1/flash.c](#)
- [ARMCM4_S32K14/flash.c](#)
- [ARMCM4_STM32F3/flash.c](#)
- [ARMCM4_STM32F4/flash.c](#)
- [ARMCM4_STM32G4/flash.c](#)
- [ARMCM4_STM32L4/flash.c](#)
- [ARMCM4_TM4C/flash.c](#)
- [ARMCM4_XMC4/flash.c](#)
- [ARMCM7_STM32F7/flash.c](#)
- [ARMCM7_STM32H7/flash.c](#)
- [HCS12/flash.c](#)
- [flash_ecc.c](#)
- [TRICORE_TC2/flash.c](#)
- [TRICORE_TC3/flash.c](#)

6.10 tFlashPrescalerSysclockMapping Struct Reference

Mapping table for finding the corect flash clock divider prescaler.

Data Fields

- `blt_int16u sysclock_min`
- `blt_int16u sysclock_max`
- `blt_int8u prescaler`

6.10.1 Detailed Description

Mapping table for finding the corect flash clock divider prescaler.

6.10.2 Field Documentation

6.10.2.1 prescaler

`blt_int8u prescaler`

prescaler for this busclock range

Referenced by [FlashInit\(\)](#).

6.10.2.2 sysclock_max

`blt_int16u sysclock_max`

max busclock for this prescaler

Referenced by [FlashInit\(\)](#).

6.10.2.3 sysclock_min

`blt_int16u sysclock_min`

min busclock for this prescaler

The documentation for this struct was generated from the following file:

- [flash_ecc.c](#)

6.11 tFlashRegs Struct Reference

Structure type for the flash control registers.

Data Fields

- volatile blt_int8u fclkdiv
- volatile blt_int8u fsec
- volatile blt_int8u ftstmod
- volatile blt_int8u fcngf
- volatile blt_int8u fprot
- volatile blt_int8u fstat
- volatile blt_int8u fcnd
- volatile blt_int8u fccobix
- volatile blt_int8u frsv0
- volatile blt_int8u fercnfg
- volatile blt_int8u ferstat
- volatile blt_int8u dfprot
- volatile blt_int16u fccob
- volatile blt_int8u frsv1
- volatile blt_int8u frsv2
- volatile blt_int8u frsv3
- volatile blt_int8u frsv4
- volatile blt_int8u fopt
- volatile blt_int8u frsv5
- volatile blt_int8u frsv6
- volatile blt_int8u frsv7

6.11.1 Detailed Description

Structure type for the flash control registers.

6.11.2 Field Documentation

6.11.2.1 dfprot

volatile blt_int8u dfprot

data flash protection register

6.11.2.2 fccob

volatile blt_int16u fccob

flash command common object reg.

6.11.2.3 fccobix

volatile `blt_int8u` fccobix

flash CCOB index register

6.11.2.4 fclkdiv

volatile `blt_int8u` fclkdiv

flash clock devider register

6.11.2.5 fcmd

volatile `blt_int8u` fcmd

flash command register

6.11.2.6 fcnfg

volatile `blt_int8u` fcnfg

flash configuration register

6.11.2.7 fercnfg

volatile `blt_int8u` fercnfg

flash error configuration reg.

6.11.2.8 ferstat

volatile `blt_int8u` ferstat

flash error status register

6.11.2.9 fopt

volatile `blt_int8u` fopt

flash option register

6.11.2.10 fprot

volatile `blt_int8u` fprot

flash protection register

program flash protection register

6.11.2.11 frsv0

volatile `blt_int8u` frsv0

flash reserver register

6.11.2.12 frsv1

volatile `blt_int8u` frsv1

flash reserver register

6.11.2.13 frsv2

volatile `blt_int8u` frsv2

flash reserver register

6.11.2.14 frsv3

volatile `blt_int8u` frsv3

flash reserver register

6.11.2.15 frsv4

volatile `blt_int8u` frsv4

flash reserver register

6.11.2.16 frsv5

```
volatile blt_int8u frsv5
```

flash reserver register

6.11.2.17 frsv6

```
volatile blt_int8u frsv6
```

flash reserver register

6.11.2.18 frsv7

```
volatile blt_int8u frsv7
```

flash reserver register

6.11.2.19 fsec

```
volatile blt_int8u fsec
```

flash security register

6.11.2.20 fstat

```
volatile blt_int8u fstat
```

flash status register

6.11.2.21 ftstmod

```
volatile blt_int8u ftstmod
```

flash test mode register

The documentation for this struct was generated from the following files:

- [HCS12/flash.c](#)
- [flash_ecc.c](#)

6.12 tFlashSector Struct Reference

Flash sector descriptor type.

Data Fields

- `blt_addr sector_start`
- `blt_int32u sector_size`
- `blt_int8u sector_num`
- `blt_int8u bank_num`

6.12.1 Detailed Description

Flash sector descriptor type.

Structure type for the flash sectors in the flash layout table.

Flash sector descriptor type. Note that in this driver the word sector is synonym to the word page, which is used in the HAL driver.

6.12.2 Field Documentation

6.12.2.1 bank_num

`blt_int8u bank_num`

bank number

Referenced by [FlashEraseSectors\(\)](#).

6.12.2.2 sector_num

`blt_int8u sector_num`

sector number

Referenced by [FlashEraseSectors\(\)](#), and [FlashGetSector\(\)](#).

6.12.2.3 sector_size

`blt_int32u sector_size`

sector size in bytes

Referenced by [FlashEmptyCheckSector\(\)](#), [FlashEraseSectors\(\)](#), and [FlashGetSectorSize\(\)](#).

6.12.2.4 sector_start

`blt_addr sector_start`

sector start address

Referenced by [FlashEmptyCheckSector\(\)](#), [FlashEraseSectors\(\)](#), [FlashGetSectorBaseAddr\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), and [FlashSwitchBlock\(\)](#).

The documentation for this struct was generated from the following files:

- [_template/flash.c](#)
- [ARMCM0_S32K11/flash.c](#)
- [ARMCM0_STM32C0/flash.c](#)
- [ARMCM0_STM32F0/flash.c](#)
- [ARMCM0_STM32G0/flash.c](#)
- [ARMCM0_STM32L0/flash.c](#)
- [ARMCM0_XMC1/flash.c](#)
- [ARMCM33_STM32H5/flash.c](#)
- [ARMCM33_STM32L5/flash.c](#)
- [ARMCM33_STM32U5/flash.c](#)
- [ARMCM3_EFM32/flash.c](#)
- [ARMCM3_LM3S/flash.c](#)
- [ARMCM3_STM32F1/flash.c](#)
- [ARMCM3_STM32F2/flash.c](#)
- [ARMCM3_STM32L1/flash.c](#)
- [ARMCM4_S32K14/flash.c](#)
- [ARMCM4_STM32F3/flash.c](#)
- [ARMCM4_STM32F4/flash.c](#)
- [ARMCM4_STM32G4/flash.c](#)
- [ARMCM4_STM32L4/flash.c](#)
- [ARMCM4_TM4C/flash.c](#)
- [ARMCM4_XMC4/flash.c](#)
- [ARMCM7_STM32F7/flash.c](#)
- [ARMCM7_STM32H7/flash.c](#)
- [HCS12/flash.c](#)
- [flash_ecc.c](#)
- [TRICORE_TC2/flash.c](#)
- [TRICORE_TC3/flash.c](#)

6.13 tSrecLineParseObject Struct Reference

Structure type for grouping the parsing results of an S-record line.

```
#include <file.h>
```

Data Fields

- `blt_char line [MAX_CHARS_PER_LINE]`
- `blt_int8u data [MAX_DATA_BYTES_PER_LINE]`
- `blt_addr address`

6.13.1 Detailed Description

Structure type for grouping the parsing results of an S-record line.

6.13.2 Field Documentation

6.13.2.1 address

`blt_addr address`

address on S1, S2 or S3 line

6.13.2.2 data

`blt_int8u data [MAX_DATA_BYTES_PER_LINE]`

array for S1, S2 or S3 data bytes

6.13.2.3 line

`blt_char line [MAX_CHARS_PER_LINE]`

string buffer for the line chars

The documentation for this struct was generated from the following file:

- `file.h`

6.14 tTimerRegs Struct Reference

Structure type with the layout of the timer related control registers.

Data Fields

- volatile blt_int8u tios
- volatile blt_int8u cforc
- volatile blt_int8u oc7m
- volatile blt_int8u oc7d
- volatile blt_int16u tcnt
- volatile blt_int8u tsr1
- volatile blt_int8u ttov
- volatile blt_int8u tctl1
- volatile blt_int8u tctl2
- volatile blt_int8u tctl3
- volatile blt_int8u tctl4
- volatile blt_int8u tie
- volatile blt_int8u tsr2
- volatile blt_int8u tf1g1
- volatile blt_int8u tf1g2
- volatile blt_int16u tc [8]

6.14.1 Detailed Description

Structure type with the layout of the timer related control registers.

6.14.2 Field Documentation

6.14.2.1 cforc

volatile blt_int8u cforc

compare force register

6.14.2.2 oc7d

volatile blt_int8u oc7d

output compare 7 data register

6.14.2.3 oc7m

```
volatile blt_int8u oc7m  
output compare 7 mask register
```

6.14.2.4 tc

```
volatile blt_int16u tc[8]  
input capture/output compare register n
```

6.14.2.5 tcnt

```
volatile blt_int16u tcnt  
timer counter register
```

6.14.2.6 tctl1

```
volatile blt_int8u tctl1  
timer control register 1
```

6.14.2.7 tctl2

```
volatile blt_int8u tctl2  
timer control register 2
```

6.14.2.8 tctl3

```
volatile blt_int8u tctl3  
timer control register 3
```

6.14.2.9 tctl4

```
volatile blt_int8u tctl4  
timer control register 4
```

6.14.2.10 tflg1

```
volatile blt_int8u tflg1  
timer interrupt flag 1
```

6.14.2.11 tflg2

```
volatile blt_int8u tflg2  
timer interrupt flag 2
```

6.14.2.12 tie

```
volatile blt_int8u tie  
interrupt enable register
```

6.14.2.13 tios

```
volatile blt_int8u tios  
input capture/output compare select
```

6.14.2.14 tscre1

```
volatile blt_int8u tscre1  
system control register 1
```

6.14.2.15 tscre2

```
volatile blt_int8u tscre2  
system control register 2
```

6.14.2.16 ttov

```
volatile blt_int8u ttov  
toggle overflow register
```

The documentation for this struct was generated from the following file:

- [HCS12/timer.c](#)

6.15 tXcpInfo Struct Reference

Structure type for grouping XCP internal module information.

Data Fields

- `blt_int8u ctoData [BOOT_COM_RX_MAX_DATA]`
- `blt_int8u connected`
- `blt_int8u protection`
- `blt_int8u s_n_k_resource`
- `blt_int8u ctoPending`
- `blt_int16s ctoLen`
- `blt_int32u mta`

6.15.1 Detailed Description

Structure type for grouping XCP internal module information.

6.15.2 Field Documentation

6.15.2.1 connected

`blt_int8u connected`

connection established

Referenced by [XcpCmdConnect\(\)](#), [XcpCmdDisconnect\(\)](#), [XcpCmdUnlock\(\)](#), [XcpInit\(\)](#), [XcpIsConnected\(\)](#), and [XcpPacketReceived\(\)](#).

6.15.2.2 ctoData

`blt_int8u ctoData[BOOT_COM_RX_MAX_DATA]`

cto packet data buffer

Referenced by [XcpCmdBuildCheckSum\(\)](#), [XcpCmdConnect\(\)](#), [XcpCmdDisconnect\(\)](#), [XcpCmdGetId\(\)](#), [XcpCmdGetSeed\(\)](#), [XcpCmdGetStatus\(\)](#), [XcpCmdProgram\(\)](#), [XcpCmdProgramClear\(\)](#), [XcpCmdProgramMax\(\)](#), [XcpCmdProgramReset\(\)](#), [XcpCmdProgramStart\(\)](#), [XcpCmdSetMta\(\)](#), [XcpCmdShortUpload\(\)](#), [XcpCmdUnlock\(\)](#), [XcpCmdUpload\(\)](#), [XcpPacketReceived\(\)](#), and [XcpSetCtoError\(\)](#).

6.15.2.3 ctoLen

`blt_int16s ctoLen`

cto current packet length

Referenced by [XcpCmdBuildCheckSum\(\)](#), [XcpCmdConnect\(\)](#), [XcpCmdDisconnect\(\)](#), [XcpCmdGetId\(\)](#), [XcpCmdGetSeed\(\)](#), [XcpCmdGetStatus\(\)](#), [XcpCmdProgram\(\)](#), [XcpCmdProgramClear\(\)](#), [XcpCmdProgramMax\(\)](#), [XcpCmdProgramReset\(\)](#), [XcpCmdProgramStart\(\)](#), [XcpCmdSetMta\(\)](#), [XcpCmdShortUpload\(\)](#), [XcpCmdUnlock\(\)](#), [XcpCmdUpload\(\)](#), [XcpInit\(\)](#), [XcpPacketReceived\(\)](#), and [XcpSetCtoError\(\)](#).

6.15.2.4 ctoPending

`blt_int8u ctoPending`

cto transmission pending flag

Referenced by [XcpInit\(\)](#), [XcpPacketReceived\(\)](#), and [XcpPacketTransmitted\(\)](#).

6.15.2.5 mta

`blt_int32u mta`

memory transfer address

Referenced by [XcpCmdBuildCheckSum\(\)](#), [XcpCmdGetId\(\)](#), [XcpCmdProgram\(\)](#), [XcpCmdProgramClear\(\)](#), [XcpCmdProgramMax\(\)](#), [XcpCmdSetMta\(\)](#), [XcpCmdShortUpload\(\)](#), [XcpCmdUpload\(\)](#), and [XcpInit\(\)](#).

6.15.2.6 protection

`blt_int8u protection`

protection state

Referenced by [XcpCmdGetSeed\(\)](#), [XcpCmdGetStatus\(\)](#), [XcpCmdProgram\(\)](#), [XcpCmdProgramClear\(\)](#), [XcpCmdProgramMax\(\)](#), [XcpCmdProgramPrepare\(\)](#), [XcpCmdProgramReset\(\)](#), [XcpCmdProgramStart\(\)](#), [XcpCmdShortUpload\(\)](#), [XcpCmdUnlock\(\)](#), [XcpCmdUpload\(\)](#), [XcpInit\(\)](#), and [XcpProtectResources\(\)](#).

6.15.2.7 s_n_k_resource

`blt_int8u s_n_k_resource`

for seed/key sequence

Referenced by [XcpCmdGetSeed\(\)](#), [XcpCmdUnlock\(\)](#), and [XcpInit\(\)](#).

The documentation for this struct was generated from the following file:

- [xcp.c](#)

6.16 uip_tcp_appstate_t Struct Reference

Define the `uip_tcp_appstate_t` datatype. This is the state of our tcp/ip application, and the memory required for this state is allocated together with each TCP connection. One application state for each TCP connection.

```
#include <net.h>
```

6.16.1 Detailed Description

Define the `uip_tcp_appstate_t` datatype. This is the state of our tcp/ip application, and the memory required for this state is allocated together with each TCP connection. One application state for each TCP connection.

The documentation for this struct was generated from the following file:

- [net.h](#)

Chapter 7

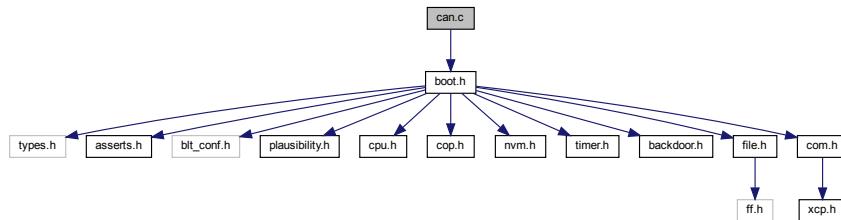
File Documentation

7.1 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
```

Include dependency graph for _template/can.c:



Data Structures

- struct [tCanBusTiming](#)
Structure type for grouping CAN bus timing related information.

Macros

- #define [CAN_MSG_TX_TIMEOUT_MS](#) (50u)
Timeout for transmitting a CAN message in milliseconds.

Functions

- static [blt_bool CanGetSpeedConfig](#) ([blt_int16u](#) baud, [blt_int16u](#) *prescaler, [blt_int8u](#) *tseg1, [blt_int8u](#) *tseg2)
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void [CanInit](#) (void)
Initializes the CAN controller and synchronizes it to the CAN bus.
- void [CanTransmitPacket](#) ([blt_int8u](#) *data, [blt_int8u](#) len)
Transmits a packet formatted for the communication interface.
- [blt_bool CanReceivePacket](#) ([blt_int8u](#) *data, [blt_int8u](#) *len)
Receives a communication interface packet if one is present.

Variables

- static const [tCanBusTiming canTiming \[\]](#)
CAN bittiming table for dynamically calculating the bittiming settings.

7.1.1 Detailed Description

Bootloader CAN communication interface source file.

7.1.2 Function Documentation

7.1.2.1 CanGetSpeedConfig()

```
static blt\_bool CanGetSpeedConfig (
    blt\_int16u baud,
    blt\_int16u * prescaler,
    blt\_int8u * tseg1,
    blt\_int8u * tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>tseg1</i> | Pointer to where the value for TSEG2 will be stored. |
| <i>tseg2</i> | Pointer to where the value for TSEG2 will be stored. |

Returns

[BLT_TRUE](#) if the CAN bustiming register values were found, [BLT_FALSE](#) otherwise.

Referenced by [CanInit\(\)](#).

7.1.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

Referenced by [ComInit\(\)](#).

7.1.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

Referenced by [ComTask\(\)](#).

7.1.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

Referenced by [ComTransmitPacket\(\)](#).

7.1.3 Variable Documentation

7.1.3.1 canTiming

```
const tCanBusTiming canTiming[] [static]
```

Initial value:

```
=
{
    { 5, 2 },
    { 6, 2 },
    { 6, 3 },
    { 7, 3 },
    { 8, 3 },
    { 9, 3 },
    { 9, 4 },
    { 10, 4 },
    { 11, 4 },
    { 12, 4 },
    { 12, 5 },
    { 13, 5 },
    { 14, 5 },
    { 15, 5 },
    { 15, 6 },
    { 16, 6 },
    { 16, 7 },
    { 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

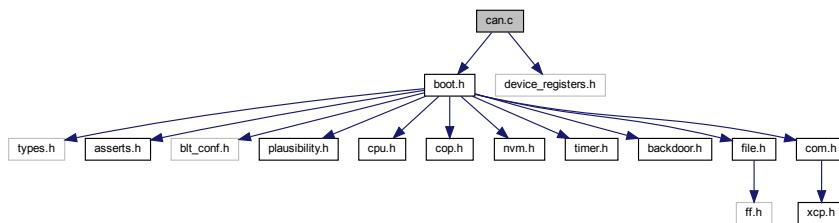
According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2) * 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

Referenced by [CanGetSpeedConfig\(\)](#).

7.2 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "device_registers.h"
Include dependency graph for ARMCM0_S32K11/can.c:
```



Data Structures

- struct [tCanBusTiming](#)

Structure type for grouping CAN bus timing related information.

Macros

- `#define CAN_INIT_TIMEOUT_MS (250U)`
Timeout for entering/leaving CAN initialization mode in milliseconds.
- `#define CAN_MSG_TX_TIMEOUT_MS (50U)`
Timeout for transmitting a CAN message in milliseconds.
- `#define CANx (CAN0)`
Set the peripheral CAN0 base pointer.
- `#define PCC_FlexCANx_INDEX (PCC_FlexCAN0_INDEX)`
Set the PCC index offset for CAN0.
- `#define CANx_MAX_MB_NUM (FEATURE_CAN0_MAX_MB_NUM)`
Set the number of message boxes supported by CAN0.
- `#define CAN_TX_MSGBOX_NUM (8U)`
The mailbox used for transmitting the XCP respond message.
- `#define CAN_RX_MSGBOX_NUM (9U)`
The mailbox used for receiving the XCP command message.

Functions

- static `blt_bool CanGetSpeedConfig (blt_int16u baud, blt_int16u *prescaler, tCanBusTiming *busTimingCfg)`
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- static void `CanFreezeModeEnter (void)`
Places the CAN controller in freeze mode. Note that the CAN controller can only be placed in freeze mode, if it is actually enabled.
- static void `CanFreezeModeExit (void)`
Leaves the CAN controller's freeze mode. Note that this operation can only be done, if it is actually enabled.
- static void `CanDisabledModeEnter (void)`
Places the CAN controller in disabled mode.
- static void `CanDisabledModeExit (void)`
Places the CAN controller in enabled mode.
- void `CanInit (void)`
Initializes the CAN controller and synchronizes it to the CAN bus.
- void `CanTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool CanReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.

Variables

- static const `tCanBusTiming canTiming []`
CAN bit timing table for dynamically calculating the bittiming settings.
- static volatile `blt_int32u dummyTimerVal`
Dummy variable to store the CAN controller's free running timer value in. This is needed at the end of a CAN message reception to unlock the mailbox again. If this variable is declared locally within the function, it generates an unwanted compiler warning about assigning a value and not using it. For this reason this dummy variable is declare here as a module global.

7.2.1 Detailed Description

Bootloader CAN communication interface source file.

7.2.2 Function Documentation

7.2.2.1 CanDisabledModeEnter()

```
static void CanDisabledModeEnter (
    void ) [static]
```

Places the CAN controller in disabled mode.

Returns

none.

Referenced by [CanInit\(\)](#).

7.2.2.2 CanDisabledModeExit()

```
static void CanDisabledModeExit (
    void ) [static]
```

Places the CAN controller in enabled mode.

Returns

none.

Referenced by [CanInit\(\)](#).

7.2.2.3 CanFreezeModeEnter()

```
static void CanFreezeModeEnter (
    void ) [static]
```

Places the CAN controller in freeze mode. Note that the CAN controller can only be placed in freeze mode, if it is actually enabled.

Returns

none.

Referenced by [CanInit\(\)](#).

7.2.2.4 CanFreezeModeExit()

```
static void CanFreezeModeExit (
    void ) [static]
```

Leaves the CAN controller's freeze mode. Note that this operation can only be done, if it is actually enabled.

Returns

none.

Referenced by [CanInit\(\)](#).

7.2.2.5 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u baud,
    blt_int16u * prescaler,
    tCanBusTiming * busTimingCfg ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|---------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>busTimingCfg</i> | Pointer to where the bus timing values will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.2.2.6 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.2.2.7 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.2.2.8 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.2.3 Variable Documentation

7.2.3.1 canTiming

```
const tCanBusTiming canTiming[ ] [static]
```

Initial value:

```
=
{
    { 8U, 3U, 2U, 2U },
```

```
{
{ 9U, 3U, 3U, 2U },
{ 10U, 3U, 3U, 3U },
{ 11U, 4U, 3U, 3U },
{ 12U, 4U, 4U, 3U },
{ 13U, 5U, 4U, 3U },
{ 14U, 5U, 4U, 4U },
{ 15U, 6U, 4U, 4U },
{ 16U, 6U, 5U, 4U },
{ 17U, 7U, 5U, 4U },
{ 18U, 7U, 5U, 5U },
{ 19U, 8U, 5U, 5U },
{ 20U, 8U, 6U, 5U },
{ 21U, 8U, 7U, 5U },
{ 22U, 8U, 7U, 6U },
{ 23U, 8U, 8U, 6U },
{ 24U, 8U, 8U, 7U },
{ 25U, 8U, 8U, 8U }
}
```

CAN bit timing table for dynamically calculating the bittiming settings.

According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + TSEG2)

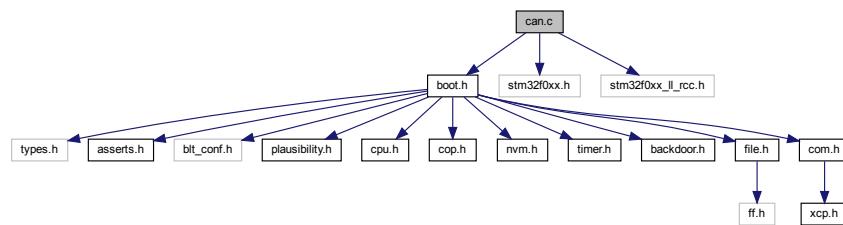
- 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%. A visual representation of the TQ in a bit is: | SYNCSEG | TIME1SEG | TIME2SEG | Or with an alternative representation: | SYNCSEG | PROPSSEG | PHASE1SEG | PHASE2SEG | With the alternative representation TIME1SEG = PROPSSEG + PHASE1SEG.

Referenced by [CanGetSpeedConfig\(\)](#).

7.3 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "stm32f0xx.h"
#include "stm32f0xx_ll_rcc.h"
Include dependency graph for ARMCM0_STM32F0/can.c:
```



Data Structures

- struct [tCanBusTiming](#)
Structure type for grouping CAN bus timing related information.

Macros

- `#define CAN_MSG_TX_TIMEOUT_MS (50u)`
Timeout for transmitting a CAN message in milliseconds.
- `#define CAN_CHANNEL CAN`
Set CAN base address to CAN1.

Functions

- static `blt_bool CanGetSpeedConfig (blt_int16u baud, blt_int16u *prescaler, blt_int8u *tseg1, blt_int8u *tseg2)`
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void `CanInit (void)`
Initializes the CAN controller and synchronizes it to the CAN bus.
- void `CanTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool CanReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.

Variables

- static const `tCanBusTiming canTiming []`
CAN bittiming table for dynamically calculating the bittiming settings.
- static CAN_HandleTypeDef `canHandle`
CAN handle to be used in API calls.

7.3.1 Detailed Description

Bootloader CAN communication interface source file.

7.3.2 Function Documentation

7.3.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u baud,
    blt_int16u * prescaler,
    blt_int8u * tseg1,
    blt_int8u * tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>tseg1</i> | Pointer to where the value for TSEG2 will be stored. |
| <i>tseg2</i> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.3.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.3.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.3.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.3.3 Variable Documentation

7.3.3.1 canTiming

```
const tCanBusTiming canTiming[ ] [static]
```

Initial value:

```
=
{
    { 5, 2 },
    { 6, 2 },
    { 6, 3 },
    { 7, 3 },
    { 8, 3 },
    { 9, 3 },
    { 9, 4 },
    { 10, 4 },
    { 11, 4 },
    { 12, 4 },
    { 12, 5 },
    { 13, 5 },
    { 14, 5 },
    { 15, 5 },
    { 15, 6 },
    { 16, 6 },
    { 16, 7 },
    { 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

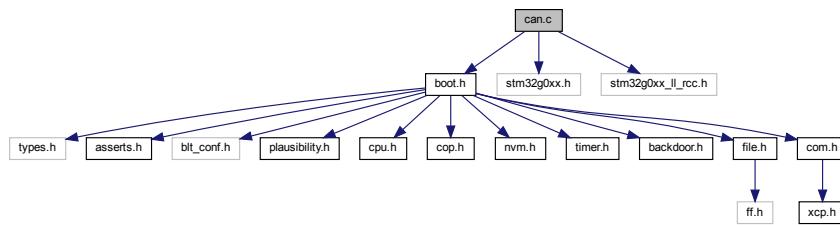
According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2) * 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

Referenced by [CanGetSpeedConfig\(\)](#).

7.4 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "stm32g0xx.h"
#include "stm32g0xx_ll_rcc.h"
Include dependency graph for ARMCM0_STM32G0/can.c:
```



Data Structures

- struct **tCanBusTiming**
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.
- #define **CAN_CHANNEL** FDCAN1
Set CAN base address to CAN1.

Functions

- static **blt_bool** CanGetSpeedConfig (**blt_int16u** baud, **blt_int16u** *prescaler, **blt_int8u** *tseg1, **blt_int8u** *tseg2)
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void **CanInit** (**void**)
Initializes the CAN controller and synchronizes it to the CAN bus.
- void **CanTransmitPacket** (**blt_int8u** *data, **blt_int8u** len)
Transmits a packet formatted for the communication interface.
- **blt_bool** **CanReceivePacket** (**blt_int8u** *data, **blt_int8u** *len)
Receives a communication interface packet if one is present.

Variables

- static const **tCanBusTiming** **canTiming** []
CAN bittiming table for dynamically calculating the bittiming settings.
- static **FDCAN_HandleTypeDef** **canHandle**
CAN handle to be used in API calls.

7.4.1 Detailed Description

Bootloader CAN communication interface source file.

7.4.2 Function Documentation

7.4.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u *baud,
    blt_int16u *prescaler,
    blt_int8u *tseg1,
    blt_int8u *tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>tseg1</i> | Pointer to where the value for TSEG2 will be stored. |
| <i>tseg2</i> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.4.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.4.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.4.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.4.3 Variable Documentation

7.4.3.1 canTiming

```
const tCanBusTiming canTiming[ ] [static]
```

Initial value:

```
=
{
    { 5, 2 },
```

```
{
{ 6, 2 },
{ 6, 3 },
{ 7, 3 },
{ 8, 3 },
{ 9, 3 },
{ 9, 4 },
{ 10, 4 },
{ 11, 4 },
{ 12, 4 },
{ 12, 5 },
{ 13, 5 },
{ 14, 5 },
{ 15, 5 },
{ 15, 6 },
{ 16, 6 },
{ 16, 7 },
{ 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2) * 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

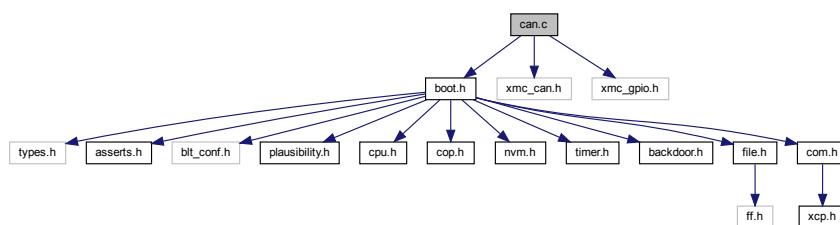
Referenced by [CanGetSpeedConfig\(\)](#).

7.5 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "xmc_can.h"
#include "xmc_gpio.h"
```

Include dependency graph for ARMCM0_XMC1/can.c:



Macros

- `#define CAN_MSG_TX_TIMEOUT_MS (50u)`
Timeout for transmitting a CAN message in milliseconds.
- `#define CAN_CHANNEL ((CAN_NODE_TypeDef *)(&canChannelMap[BOOT_COM_CAN_CHANNEL_<INDEX>]))`
Macro for accessing the CAN channel handle in the format that is expected by the XMClib CAN driver.
- `#define CAN_TX_MSBOBJ (CAN_MO0)`
Message object dedicated to message transmission.
- `#define CAN_TX_MSBOBJ_IDX (0)`
Index of the message object dedicated to message transmission.
- `#define CAN_RX_MSBOBJ (CAN_MO1)`
Message object dedicated to message reception.
- `#define CAN_RX_MSBOBJ_IDX (1)`
Index of the message object dedicated to message reception.

Functions

- void **CanInit** (void)

Initializes the CAN controller and synchronizes it to the CAN bus.
- void **CanTransmitPacket** (**blt_int8u** *data, **blt_int8u** len)

Transmits a packet formatted for the communication interface.
- **blt_bool** **CanReceivePacket** (**blt_int8u** *data, **blt_int8u** *len)

Receives a communication interface packet if one is present.

Variables

- static const CAN_NODE_TypeDef * **canChannelMap** []

Helper array to quickly convert the channel index, as specific in the boot-loader's configuration header, to the associated channel handle that the XMCLib's CAN driver requires.
- static XMC_CAN_MO_t **transmitMsgObj**

Transmit message object data structure.
- static XMC_CAN_MO_t **receiveMsgObj**

Receive message object data structure.

7.5.1 Detailed Description

Bootloader CAN communication interface source file.

7.5.2 Function Documentation

7.5.2.1 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.5.2.2 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.5.2.3 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

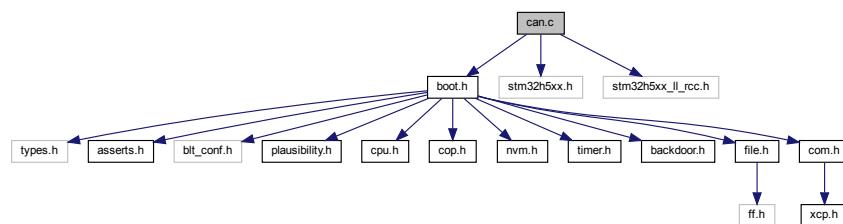
Returns

none.

7.6 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "stm32h5xx.h"
#include "stm32h5xx_ll_rcc.h"
Include dependency graph for ARMCM33_STM32H5/can.c:
```



Data Structures

- struct **tCanBusTiming**
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.
- #define **CAN_CHANNEL** FDCAN1
Set CAN base address to CAN1.

Functions

- static **blt_bool CanGetSpeedConfig (blt_int16u baud, blt_int16u *prescaler, blt_int8u *tseg1, blt_int8u *tseg2)**
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void **CanInit (void)**
Initializes the CAN controller and synchronizes it to the CAN bus.
- void **CanTransmitPacket (blt_int8u *data, blt_int8u len)**
Transmits a packet formatted for the communication interface.
- **blt_bool CanReceivePacket (blt_int8u *data, blt_int8u *len)**
Receives a communication interface packet if one is present.

Variables

- static const **tCanBusTiming canTiming []**
CAN bittiming table for dynamically calculating the bittiming settings.
- static FDCAN_HandleTypeDef **canHandle**
CAN handle to be used in API calls.

7.6.1 Detailed Description

Bootloader CAN communication interface source file.

7.6.2 Function Documentation

7.6.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u baud,
    blt_int16u * prescaler,
    blt_int8u * tseg1,
    blt_int8u * tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>tseg1</i> | Pointer to where the value for TSEG2 will be stored. |
| <i>tseg2</i> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.6.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.6.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.6.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.6.3 Variable Documentation

7.6.3.1 canTiming

```
const tCanBusTiming canTiming[ ] [static]
```

Initial value:

```
=
{
    { 5, 2 },
    { 6, 2 },
    { 6, 3 },
    { 7, 3 },
    { 8, 3 },
    { 9, 3 },
    { 9, 4 },
    { 10, 4 },
    { 11, 4 },
    { 12, 4 },
    { 12, 5 },
    { 13, 5 },
    { 14, 5 },
    { 15, 5 },
    { 15, 6 },
    { 16, 6 },
    { 16, 7 },
    { 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

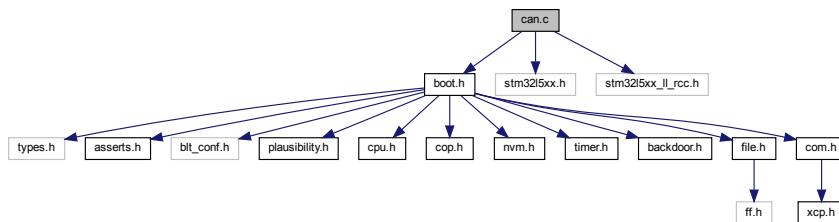
According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2) * 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

Referenced by [CanGetSpeedConfig\(\)](#).

7.7 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "stm32l5xx.h"
#include "stm32l5xx_ll_rcc.h"
Include dependency graph for ARMCM33_STM32L5/can.c:
```



Data Structures

- struct **tCanBusTiming**
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.
- #define **CAN_CHANNEL** FDCAN1
Set CAN base address to CAN1.

Functions

- static **blt_bool** CanGetSpeedConfig (**blt_int16u** baud, **blt_int16u** *prescaler, **blt_int8u** *tseg1, **blt_int8u** *tseg2)
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void **CanInit** (void)
Initializes the CAN controller and synchronizes it to the CAN bus.
- void **CanTransmitPacket** (**blt_int8u** *data, **blt_int8u** len)
Transmits a packet formatted for the communication interface.
- **blt_bool** **CanReceivePacket** (**blt_int8u** *data, **blt_int8u** *len)
Receives a communication interface packet if one is present.

Variables

- static const **tCanBusTiming** **canTiming** []
CAN bittiming table for dynamically calculating the bittiming settings.
- static **FDCAN_HandleTypeDef** **canHandle**
CAN handle to be used in API calls.

7.7.1 Detailed Description

Bootloader CAN communication interface source file.

7.7.2 Function Documentation

7.7.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u *baud,
    blt_int16u *prescaler,
    blt_int8u *tseg1,
    blt_int8u *tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>tseg1</i> | Pointer to where the value for TSEG2 will be stored. |
| <i>tseg2</i> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.7.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.7.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.7.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.7.3 Variable Documentation

7.7.3.1 canTiming

```
const tCanBusTiming canTiming[ ] [static]
```

Initial value:

```
=
{
    { 5, 2 },
```

```
{
{ 6, 2 },
{ 6, 3 },
{ 7, 3 },
{ 8, 3 },
{ 9, 3 },
{ 9, 4 },
{ 10, 4 },
{ 11, 4 },
{ 12, 4 },
{ 12, 5 },
{ 13, 5 },
{ 14, 5 },
{ 15, 5 },
{ 15, 6 },
{ 16, 6 },
{ 16, 7 },
{ 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

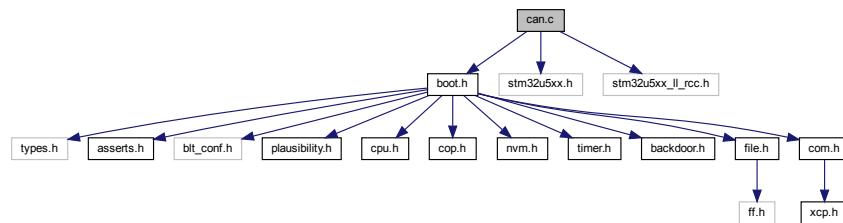
According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2) * 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

Referenced by [CanGetSpeedConfig\(\)](#).

7.8 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "stm32u5xx.h"
#include "stm32u5xx_ll_rcc.h"
Include dependency graph for ARMCM33_STM32U5/can.c:
```



Data Structures

- struct [tCanBusTiming](#)
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.
- #define **CAN_CHANNEL** FDCAN1
Set CAN base address to CAN1.

Functions

- static `blt_bool CanGetSpeedConfig (blt_int16u baud, blt_int16u *prescaler, blt_int8u *tseg1, blt_int8u *tseg2)`
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void `CanInit (void)`
Initializes the CAN controller and synchronizes it to the CAN bus.
- void `CanTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool CanReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.

Variables

- static const `tCanBusTiming canTiming []`
CAN bittiming table for dynamically calculating the bittiming settings.
- static `FDCAN_HandleTypeDef canHandle`
CAN handle to be used in API calls.

7.8.1 Detailed Description

Bootloader CAN communication interface source file.

7.8.2 Function Documentation

7.8.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u baud,
    blt_int16u * prescaler,
    blt_int8u * tseg1,
    blt_int8u * tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------------|--|
| <code>baud</code> | The desired baudrate in kbps. Valid values are 10..1000. |
| <code>prescaler</code> | Pointer to where the value for the prescaler will be stored. |
| <code>tseg1</code> | Pointer to where the value for TSEG2 will be stored. |
| <code>tseg2</code> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.8.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.8.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.8.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.8.3 Variable Documentation

7.8.3.1 canTiming

```
const tCanBusTiming canTiming[] [static]
```

Initial value:

```
=
{
    { 5, 2 },
    { 6, 2 },
    { 6, 3 },
    { 7, 3 },
    { 8, 3 },
    { 9, 3 },
    { 9, 4 },
    { 10, 4 },
    { 11, 4 },
    { 12, 4 },
    { 12, 5 },
    { 13, 5 },
    { 14, 5 },
    { 15, 5 },
    { 15, 6 },
    { 16, 6 },
    { 16, 7 },
    { 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2) * 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

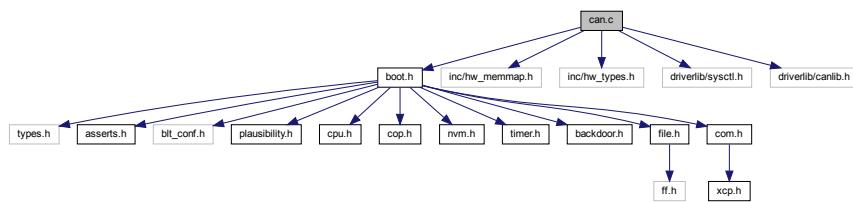
Referenced by [CanGetSpeedConfig\(\)](#).

7.9 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/canlib.h"

Include dependency graph for ARMCM3_LM3S/can.c:
```



Macros

- `#define CAN_MSG_TX_TIMEOUT_MS (50u)`
Timeout for transmitting a CAN message in milliseconds.
- `#define CAN_RX_MSGOBJECT_IDX (0)`
Index of the used reception message object.
- `#define CAN_TX_MSGOBJECT_IDX (1)`
Index of the used transmission message object.

Functions

- static `blt_int8u CanSetBittiming (void)`
*Attempts to match the bittiming parameters to the requested baudrate for a sample point between 65 and 75%, through a linear search algorithm. It is based on the equation: baudrate = CAN Clock Freq/((1+PropSeg+Phase1+Seg+Phase2Seg)*Prescaler)*
- void `CanInit (void)`
Initializes the CAN controller and synchronizes it to the CAN bus.
- void `CanTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool CanReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.

Variables

- static const `blt_int16u canBitNum2Mask []`
Lookup table to quickly and efficiently convert a bit number to a bit mask.

7.9.1 Detailed Description

Bootloader CAN communication interface source file.

7.9.2 Function Documentation

7.9.2.1 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.9.2.2 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.9.2.3 CanSetBittiming()

```
static blt_int8u CanSetBittiming (
    void ) [static]
```

Attempts to match the bittiming parameters to the requested baudrate for a sample point between 65 and 75%, through a linear search algorithm. It is based on the equation: baudrate = CAN Clock Freq/((1+PropSeg+Phase1Seg+Phase2Seg)*Prescaler)

Returns

`BLT_TRUE` if a valid bittiming configuration was found and set. `BLT_FALSE` otherwise.

Referenced by [CanInit\(\)](#).

7.9.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------------|--|
| <code>data</code> | Pointer to byte array with data that it to be transmitted. |
| <code>len</code> | Number of bytes that are to be transmitted. |

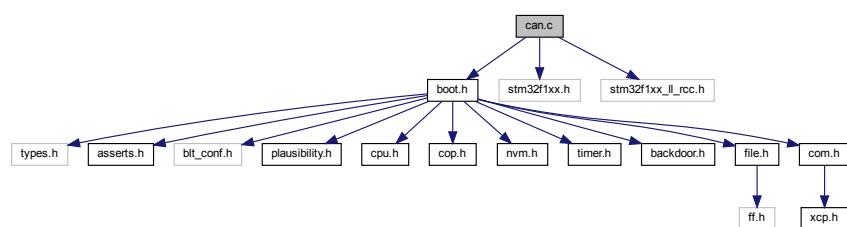
Returns

none.

7.10 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "stm32f1xx.h"
#include "stm32f1xx_ll_rcc.h"
Include dependency graph for ARMCM3_STM32F1/can.c:
```



Data Structures

- struct **tCanBusTiming**
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.
- #define **CAN_CHANNEL** CAN1
Set CAN base address to CAN1.

Functions

- static **blt_bool CanGetSpeedConfig (blt_int16u baud, blt_int16u *prescaler, blt_int8u *tseg1, blt_int8u *tseg2)**
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void **CanInit (void)**
Initializes the CAN controller and synchronizes it to the CAN bus.
- void **CanTransmitPacket (blt_int8u *data, blt_int8u len)**
Transmits a packet formatted for the communication interface.
- **blt_bool CanReceivePacket (blt_int8u *data, blt_int8u *len)**
Receives a communication interface packet if one is present.

Variables

- static const **tCanBusTiming canTiming []**
CAN bittiming table for dynamically calculating the bittiming settings.
- static CAN_HandleTypeDef **canHandle**
CAN handle to be used in API calls.

7.10.1 Detailed Description

Bootloader CAN communication interface source file.

7.10.2 Function Documentation

7.10.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u baud,
    blt_int16u * prescaler,
    blt_int8u * tseg1,
    blt_int8u * tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>tseg1</i> | Pointer to where the value for TSEG2 will be stored. |
| <i>tseg2</i> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.10.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.10.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.10.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.10.3 Variable Documentation

7.10.3.1 canTiming

```
const tCanBusTiming canTiming[ ] [static]
```

Initial value:

```
=
{
    { 5, 2 },
    { 6, 2 },
    { 6, 3 },
    { 7, 3 },
    { 8, 3 },
    { 9, 3 },
    { 9, 4 },
    { 10, 4 },
    { 11, 4 },
    { 12, 4 },
    { 12, 5 },
    { 13, 5 },
    { 14, 5 },
    { 15, 5 },
    { 15, 6 },
    { 16, 6 },
    { 16, 7 },
    { 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2) * 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

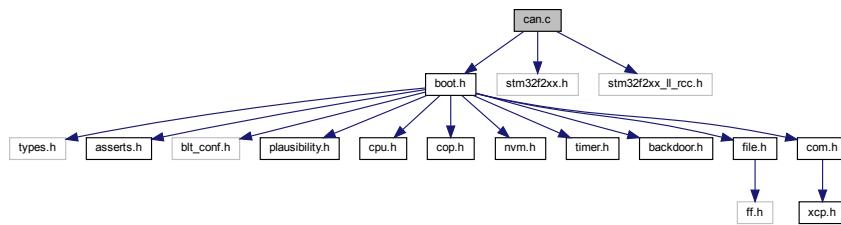
Referenced by [CanGetSpeedConfig\(\)](#).

7.11 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "stm32f2xx.h"
#include "stm32f2xx_ll_rcc.h"

Include dependency graph for ARMCM3_STM32F2/can.c:
```



Data Structures

- struct **tCanBusTiming**
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.
- #define **CAN_CHANNEL** CAN1
Set CAN base address to CAN1.

Functions

- static **blt_bool** CanGetSpeedConfig (**blt_int16u** baud, **blt_int16u** *prescaler, **blt_int8u** *tseg1, **blt_int8u** *tseg2)
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void **CanInit** (void)
Initializes the CAN controller and synchronizes it to the CAN bus.
- void **CanTransmitPacket** (**blt_int8u** *data, **blt_int8u** len)
Transmits a packet formatted for the communication interface.
- **blt_bool** **CanReceivePacket** (**blt_int8u** *data, **blt_int8u** *len)
Receives a communication interface packet if one is present.

Variables

- static const **tCanBusTiming** **canTiming** []
CAN bittiming table for dynamically calculating the bittiming settings.
- static **CAN_HandleTypeDef** **canHandle**
CAN handle to be used in API calls.

7.11.1 Detailed Description

Bootloader CAN communication interface source file.

7.11.2 Function Documentation

7.11.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u *baud,
    blt_int16u *prescaler,
    blt_int8u *tseg1,
    blt_int8u *tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>tseg1</i> | Pointer to where the value for TSEG2 will be stored. |
| <i>tseg2</i> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.11.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.11.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.11.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.11.3 Variable Documentation

7.11.3.1 canTiming

```
const tCanBusTiming canTiming[ ] [static]
```

Initial value:

```
=
{
    { 5, 2 },
```

```
{
{ 6, 2 },
{ 6, 3 },
{ 7, 3 },
{ 8, 3 },
{ 9, 3 },
{ 9, 4 },
{ 10, 4 },
{ 11, 4 },
{ 12, 4 },
{ 12, 5 },
{ 13, 5 },
{ 14, 5 },
{ 15, 5 },
{ 15, 6 },
{ 16, 6 },
{ 16, 7 },
{ 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

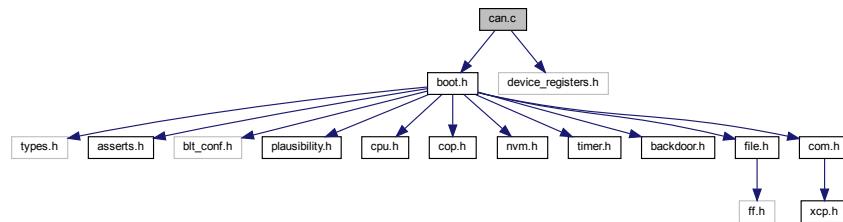
According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2) * 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

Referenced by [CanGetSpeedConfig\(\)](#).

7.12 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "device_registers.h"
Include dependency graph for ARMCM4_S32K14/can.c:
```



Data Structures

- struct [tCanBusTiming](#)
Structure type for grouping CAN bus timing related information.

Macros

- `#define CAN_INIT_TIMEOUT_MS (250U)`
Timeout for entering/leaving CAN initialization mode in milliseconds.
- `#define CAN_MSG_TX_TIMEOUT_MS (50U)`
Timeout for transmitting a CAN message in milliseconds.
- `#define CANx (CAN0)`
Set the peripheral CAN0 base pointer.
- `#define PCC_FlexCANx_INDEX (PCC_FlexCAN0_INDEX)`
Set the PCC index offset for CAN0.
- `#define CANx_MAX_MB_NUM (FEATURE_CAN0_MAX_MB_NUM)`
Set the number of message boxes supported by CAN0.
- `#define CAN_TX_MSGBOX_NUM (8U)`
The mailbox used for transmitting the XCP respond message.
- `#define CAN_RX_MSGBOX_NUM (9U)`
The mailbox used for receiving the XCP command message.

Functions

- static `blt_bool CanGetSpeedConfig (blt_int16u baud, blt_int16u *prescaler, tCanBusTiming *busTimingCfg)`
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- static void `CanFreezeModeEnter (void)`
Places the CAN controller in freeze mode. Note that the CAN controller can only be placed in freeze mode, if it is actually enabled.
- static void `CanFreezeModeExit (void)`
Leaves the CAN controller's freeze mode. Note that this operation can only be done, if it is actually enabled.
- static void `CanDisabledModeEnter (void)`
Places the CAN controller in disabled mode.
- static void `CanDisabledModeExit (void)`
Places the CAN controller in enabled mode.
- void `CanInit (void)`
Initializes the CAN controller and synchronizes it to the CAN bus.
- void `CanTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool CanReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.

Variables

- static const `tCanBusTiming canTiming []`
CAN bit timing table for dynamically calculating the bittiming settings.
- static volatile `blt_int32u dummyTimerVal`
Dummy variable to store the CAN controller's free running timer value in. This is needed at the end of a CAN message reception to unlock the mailbox again. If this variable is declared locally within the function, it generates an unwanted compiler warning about assigning a value and not using it. For this reason this dummy variable is declare here as a module global.

7.12.1 Detailed Description

Bootloader CAN communication interface source file.

7.12.2 Function Documentation

7.12.2.1 CanDisabledModeEnter()

```
static void CanDisabledModeEnter (
    void ) [static]
```

Places the CAN controller in disabled mode.

Returns

none.

Referenced by [CanInit\(\)](#).

7.12.2.2 CanDisabledModeExit()

```
static void CanDisabledModeExit (
    void ) [static]
```

Places the CAN controller in enabled mode.

Returns

none.

Referenced by [CanInit\(\)](#).

7.12.2.3 CanFreezeModeEnter()

```
static void CanFreezeModeEnter (
    void ) [static]
```

Places the CAN controller in freeze mode. Note that the CAN controller can only be placed in freeze mode, if it is actually enabled.

Returns

none.

Referenced by [CanInit\(\)](#).

7.12.2.4 CanFreezeModeExit()

```
static void CanFreezeModeExit (
    void ) [static]
```

Leaves the CAN controller's freeze mode. Note that this operation can only be done, if it is actually enabled.

Returns

none.

Referenced by [CanInit\(\)](#).

7.12.2.5 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u baud,
    blt_int16u * prescaler,
    tCanBusTiming * busTimingCfg ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|---------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>busTimingCfg</i> | Pointer to where the bus timing values will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.12.2.6 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.12.2.7 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.12.2.8 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.12.3 Variable Documentation

7.12.3.1 canTiming

```
const tCanBusTiming canTiming[ ] [static]
```

Initial value:

```
=
{
    { 8U, 3U, 2U, 2U },
```

```
{
{ 9U, 3U, 3U, 2U },
{ 10U, 3U, 3U, 3U },
{ 11U, 4U, 3U, 3U },
{ 12U, 4U, 4U, 3U },
{ 13U, 5U, 4U, 3U },
{ 14U, 5U, 4U, 4U },
{ 15U, 6U, 4U, 4U },
{ 16U, 6U, 5U, 4U },
{ 17U, 7U, 5U, 4U },
{ 18U, 7U, 5U, 5U },
{ 19U, 8U, 5U, 5U },
{ 20U, 8U, 6U, 5U },
{ 21U, 8U, 7U, 5U },
{ 22U, 8U, 7U, 6U },
{ 23U, 8U, 8U, 6U },
{ 24U, 8U, 8U, 7U },
{ 25U, 8U, 8U, 8U }
}
```

CAN bit timing table for dynamically calculating the bittiming settings.

According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is $(\text{SYNC} + \text{TSEG1}) / (\text{SYNC} + \text{TSEG1} + \text{TSEG2})$

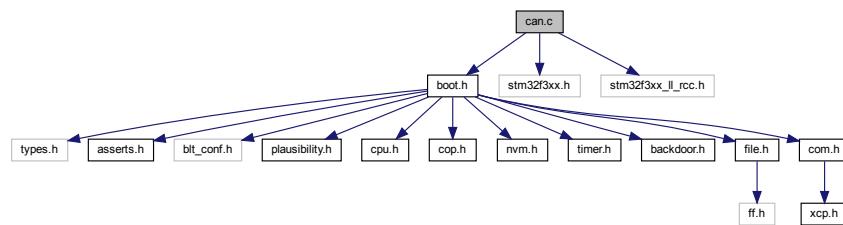
- 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%. A visual representation of the TQ in a bit is: | SYNCSEG | TIME1SEG | TIME2SEG | Or with an alternative representation: | SYNCSEG | PROPSSEG | PHASE1SEG | PHASE2SEG | With the alternative representation TIME1SEG = PROPSSEG + PHASE1SEG.

Referenced by [CanGetSpeedConfig\(\)](#).

7.13 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "stm32f3xx.h"
#include "stm32f3xx_ll_rcc.h"
Include dependency graph for ARMCM4_STM32F3/can.c:
```



Data Structures

- struct [tCanBusTiming](#)
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.
- #define **CAN_CHANNEL CAN**
Set CAN base address to CAN1.

Functions

- static **blt_bool CanGetSpeedConfig (blt_int16u baud, blt_int16u *prescaler, blt_int8u *tseg1, blt_int8u *tseg2)**
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void **CanInit (void)**
Initializes the CAN controller and synchronizes it to the CAN bus.
- void **CanTransmitPacket (blt_int8u *data, blt_int8u len)**
Transmits a packet formatted for the communication interface.
- **blt_bool CanReceivePacket (blt_int8u *data, blt_int8u *len)**
Receives a communication interface packet if one is present.

Variables

- static const **tCanBusTiming canTiming []**
CAN bittiming table for dynamically calculating the bittiming settings.
- static **CAN_HandleTypeDef canHandle**
CAN handle to be used in API calls.

7.13.1 Detailed Description

Bootloader CAN communication interface source file.

7.13.2 Function Documentation

7.13.2.1 **CanGetSpeedConfig()**

```
static blt_bool CanGetSpeedConfig (
    blt_int16u baud,
    blt_int16u * prescaler,
    blt_int8u * tseg1,
    blt_int8u * tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>tseg1</i> | Pointer to where the value for TSEG2 will be stored. |
| <i>tseg2</i> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.13.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.13.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.13.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.13.3 Variable Documentation

7.13.3.1 canTiming

```
const tCanBusTiming canTiming[ ] [static]
```

Initial value:

```
=
{
    { 5, 2 },
    { 6, 2 },
    { 6, 3 },
    { 7, 3 },
    { 8, 3 },
    { 9, 3 },
    { 9, 4 },
    { 10, 4 },
    { 11, 4 },
    { 12, 4 },
    { 12, 5 },
    { 13, 5 },
    { 14, 5 },
    { 15, 5 },
    { 15, 6 },
    { 16, 6 },
    { 16, 7 },
    { 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2) * 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

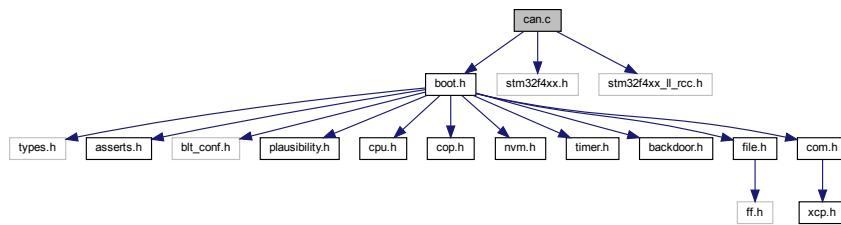
Referenced by [CanGetSpeedConfig\(\)](#).

7.14 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "stm32f4xx.h"
#include "stm32f4xx_ll_rcc.h"

Include dependency graph for ARMCM4_STM32F4/can.c:
```



Data Structures

- struct **tCanBusTiming**
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.
- #define **CAN_CHANNEL** CAN1
Set CAN base address to CAN1.

Functions

- static **blt_bool** CanGetSpeedConfig (**blt_int16u** baud, **blt_int16u** *prescaler, **blt_int8u** *tseg1, **blt_int8u** *tseg2)
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void **CanInit** (void)
Initializes the CAN controller and synchronizes it to the CAN bus.
- void **CanTransmitPacket** (**blt_int8u** *data, **blt_int8u** len)
Transmits a packet formatted for the communication interface.
- **blt_bool** **CanReceivePacket** (**blt_int8u** *data, **blt_int8u** *len)
Receives a communication interface packet if one is present.

Variables

- static const **tCanBusTiming** **canTiming** []
CAN bitTiming table for dynamically calculating the bitTiming settings.
- static **CAN_HandleTypeDef** **canHandle**
CAN handle to be used in API calls.

7.14.1 Detailed Description

Bootloader CAN communication interface source file.

7.14.2 Function Documentation

7.14.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u *baud,
    blt_int16u *prescaler,
    blt_int8u *tseg1,
    blt_int8u *tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>tseg1</i> | Pointer to where the value for TSEG2 will be stored. |
| <i>tseg2</i> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.14.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.14.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.14.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.14.3 Variable Documentation

7.14.3.1 canTiming

```
const tCanBusTiming canTiming[ ] [static]
```

Initial value:

```
=
{
    { 5, 2 },
```

```
{
{ 6, 2 },
{ 6, 3 },
{ 7, 3 },
{ 8, 3 },
{ 9, 3 },
{ 9, 4 },
{ 10, 4 },
{ 11, 4 },
{ 12, 4 },
{ 12, 5 },
{ 13, 5 },
{ 14, 5 },
{ 15, 5 },
{ 15, 6 },
{ 16, 6 },
{ 16, 7 },
{ 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

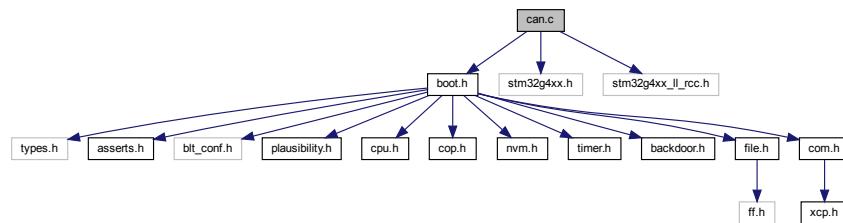
According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2) * 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

Referenced by [CanGetSpeedConfig\(\)](#).

7.15 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "stm32g4xx.h"
#include "stm32g4xx_ll_rcc.h"
Include dependency graph for ARMCM4_STM32G4/can.c:
```



Data Structures

- struct [tCanBusTiming](#)
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.
- #define **CAN_CHANNEL** FDCAN1
Set CAN base address to CAN1.

Functions

- static `blt_bool CanGetSpeedConfig (blt_int16u baud, blt_int16u *prescaler, blt_int8u *tseg1, blt_int8u *tseg2)`
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void `CanInit (void)`
Initializes the CAN controller and synchronizes it to the CAN bus.
- void `CanTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool CanReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.

Variables

- static const `tCanBusTiming canTiming []`
CAN bittiming table for dynamically calculating the bittiming settings.
- static `FDCAN_HandleTypeDef canHandle`
CAN handle to be used in API calls.

7.15.1 Detailed Description

Bootloader CAN communication interface source file.

7.15.2 Function Documentation

7.15.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u baud,
    blt_int16u * prescaler,
    blt_int8u * tseg1,
    blt_int8u * tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------------|--|
| <code>baud</code> | The desired baudrate in kbps. Valid values are 10..1000. |
| <code>prescaler</code> | Pointer to where the value for the prescaler will be stored. |
| <code>tseg1</code> | Pointer to where the value for TSEG2 will be stored. |
| <code>tseg2</code> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.15.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.15.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.15.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.15.3 Variable Documentation

7.15.3.1 canTiming

```
const tCanBusTiming canTiming[] [static]
```

Initial value:

```
=
{
    { 5, 2 },
    { 6, 2 },
    { 6, 3 },
    { 7, 3 },
    { 8, 3 },
    { 9, 3 },
    { 9, 4 },
    { 10, 4 },
    { 11, 4 },
    { 12, 4 },
    { 12, 5 },
    { 13, 5 },
    { 14, 5 },
    { 15, 5 },
    { 15, 6 },
    { 16, 6 },
    { 16, 7 },
    { 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

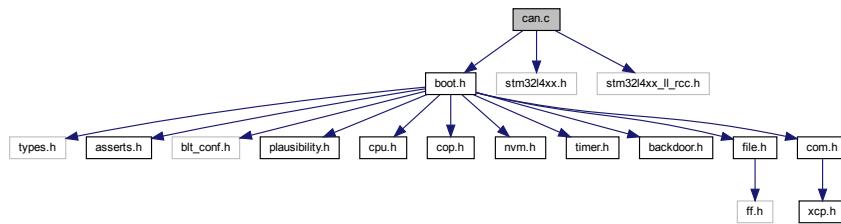
According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is $(\text{SYNC} + \text{TSEG1}) / (\text{SYNC} + \text{TSEG1} + \text{TSEG2}) * 100\%$. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

Referenced by [CanGetSpeedConfig\(\)](#).

7.16 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "stm3214xx.h"
#include "stm3214xx_ll_rcc.h"
Include dependency graph for ARMCM4_STM32L4/can.c:
```



Data Structures

- struct **tCanBusTiming**
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.

Functions

- static **blt_bool** CanGetSpeedConfig (**blt_int16u** baud, **blt_int16u** *prescaler, **blt_int8u** *tseg1, **blt_int8u** *tseg2)
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void **CanInit** (void)
Initializes the CAN controller and synchronizes it to the CAN bus.
- void **CanTransmitPacket** (**blt_int8u** *data, **blt_int8u** len)
Transmits a packet formatted for the communication interface.
- **blt_bool** **CanReceivePacket** (**blt_int8u** *data, **blt_int8u** *len)
Receives a communication interface packet if one is present.

Variables

- static const **tCanBusTiming** **canTiming** []
CAN bittiming table for dynamically calculating the bittiming settings.
- static **CAN_HandleTypeDef** **canHandle**
CAN handle to be used in API calls.

7.16.1 Detailed Description

Bootloader CAN communication interface source file.

7.16.2 Function Documentation

7.16.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u *baud,
    blt_int16u *prescaler,
    blt_int8u *tseg1,
    blt_int8u *tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>tseg1</i> | Pointer to where the value for TSEG2 will be stored. |
| <i>tseg2</i> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.16.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.16.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.16.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.16.3 Variable Documentation

7.16.3.1 canTiming

```
const tCanBusTiming canTiming[ ] [static]
```

Initial value:

```
=
{
    { 5, 2 },
```

```
{
{ 6, 2 },
{ 6, 3 },
{ 7, 3 },
{ 8, 3 },
{ 9, 3 },
{ 9, 4 },
{ 10, 4 },
{ 11, 4 },
{ 12, 4 },
{ 12, 5 },
{ 13, 5 },
{ 14, 5 },
{ 15, 5 },
{ 15, 6 },
{ 16, 6 },
{ 16, 7 },
{ 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2) * 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

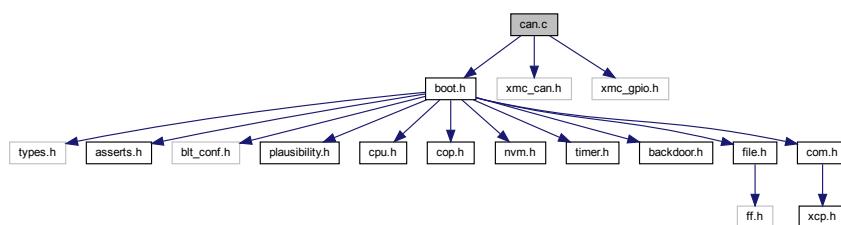
Referenced by [CanGetSpeedConfig\(\)](#).

7.17 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "xmc_can.h"
#include "xmc_gpio.h"
```

Include dependency graph for ARMCM4_XMC4/can.c:



Macros

- **#define CAN_MSG_TX_TIMEOUT_MS (50u)**
Timeout for transmitting a CAN message in milliseconds.
- **#define CAN_CHANNEL ((CAN_NODE_TypeDef *)(&canChannelMap[BOOT_COM_CAN_CHANNEL_<INDEX>]))**
Macro for accessing the CAN channel handle in the format that is expected by the XMCLib CAN driver.
- **#define CAN_TX_MSBOBJ (CAN_MO0)**
Message object dedicated to message transmission.
- **#define CAN_TX_MSBOBJ_IDX (0)**
Index of the message object dedicated to message transmission.
- **#define CAN_RX_MSBOBJ (CAN_MO1)**
Message object dedicated to message reception.
- **#define CAN_RX_MSBOBJ_IDX (1)**
Index of the message object dedicated to message reception.

Functions

- void **CanInit** (void)

Initializes the CAN controller and synchronizes it to the CAN bus.
- void **CanTransmitPacket** (**blt_int8u** *data, **blt_int8u** len)

Transmits a packet formatted for the communication interface.
- **blt_bool** **CanReceivePacket** (**blt_int8u** *data, **blt_int8u** *len)

Receives a communication interface packet if one is present.

Variables

- static const CAN_NODE_TypeDef * **canChannelMap** []

Helper array to quickly convert the channel index, as specific in the boot-loader's configuration header, to the associated channel handle that the XMCLib's CAN driver requires.
- static XMC_CAN_MO_t **transmitMsgObj**

Transmit message object data structure.
- static XMC_CAN_MO_t **receiveMsgObj**

Receive message object data structure.

7.17.1 Detailed Description

Bootloader CAN communication interface source file.

7.17.2 Function Documentation

7.17.2.1 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.17.2.2 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.17.2.3 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

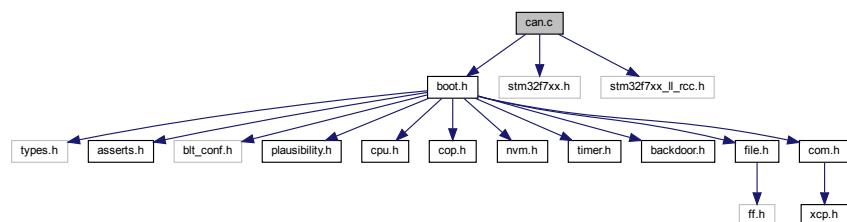
Returns

none.

7.18 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "stm32f7xx.h"
#include "stm32f7xx_ll_rcc.h"
Include dependency graph for ARMCM7_STM32F7/can.c:
```



Data Structures

- struct **tCanBusTiming**
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.
- #define **CAN_CHANNEL** CAN1
Set CAN base address to CAN1.

Functions

- static **blt_bool CanGetSpeedConfig (blt_int16u baud, blt_int16u *prescaler, blt_int8u *tseg1, blt_int8u *tseg2)**
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void **CanInit (void)**
Initializes the CAN controller and synchronizes it to the CAN bus.
- void **CanTransmitPacket (blt_int8u *data, blt_int8u len)**
Transmits a packet formatted for the communication interface.
- **blt_bool CanReceivePacket (blt_int8u *data, blt_int8u *len)**
Receives a communication interface packet if one is present.

Variables

- static const **tCanBusTiming canTiming []**
CAN bittiming table for dynamically calculating the bittiming settings.
- static CAN_HandleTypeDef **canHandle**
CAN handle to be used in API calls.

7.18.1 Detailed Description

Bootloader CAN communication interface source file.

7.18.2 Function Documentation

7.18.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u baud,
    blt_int16u * prescaler,
    blt_int8u * tseg1,
    blt_int8u * tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>tseg1</i> | Pointer to where the value for TSEG2 will be stored. |
| <i>tseg2</i> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.18.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.18.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.18.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.18.3 Variable Documentation

7.18.3.1 canTiming

```
const tCanBusTiming canTiming[ ] [static]
```

Initial value:

```
=
{
    { 5, 2 },
    { 6, 2 },
    { 6, 3 },
    { 7, 3 },
    { 8, 3 },
    { 9, 3 },
    { 9, 4 },
    { 10, 4 },
    { 11, 4 },
    { 12, 4 },
    { 12, 5 },
    { 13, 5 },
    { 14, 5 },
    { 15, 5 },
    { 15, 6 },
    { 16, 6 },
    { 16, 7 },
    { 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

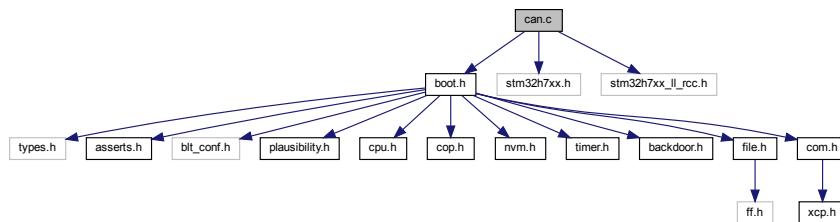
According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2) * 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

Referenced by [CanGetSpeedConfig\(\)](#).

7.19 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "stm32h7xx.h"
#include "stm32h7xx_ll_rcc.h"
Include dependency graph for ARMCM7_STM32H7/can.c:
```



Data Structures

- struct **tCanBusTiming**
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.
- #define **CAN_CHANNEL** FDCAN1
Set CAN base address to CAN1.

Functions

- static **blt_bool** CanGetSpeedConfig (**blt_int16u** baud, **blt_int16u** *prescaler, **blt_int8u** *tseg1, **blt_int8u** *tseg2)
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void **CanInit** (void)
Initializes the CAN controller and synchronizes it to the CAN bus.
- void **CanTransmitPacket** (**blt_int8u** *data, **blt_int8u** len)
Transmits a packet formatted for the communication interface.
- **blt_bool** **CanReceivePacket** (**blt_int8u** *data, **blt_int8u** *len)
Receives a communication interface packet if one is present.

Variables

- static const **tCanBusTiming** **canTiming** []
CAN bittiming table for dynamically calculating the bittiming settings.
- static **FDCAN_HandleTypeDef** **canHandle**
CAN handle to be used in API calls.

7.19.1 Detailed Description

Bootloader CAN communication interface source file.

7.19.2 Function Documentation

7.19.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u *baud,
    blt_int16u *prescaler,
    blt_int8u *tseg1,
    blt_int8u *tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>tseg1</i> | Pointer to where the value for TSEG2 will be stored. |
| <i>tseg2</i> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.19.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.19.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.19.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.19.3 Variable Documentation

7.19.3.1 canTiming

```
const tCanBusTiming canTiming[ ] [static]
```

Initial value:

```
=
{
    { 5, 2 },
```

```
{
{ 6, 2 },
{ 6, 3 },
{ 7, 3 },
{ 8, 3 },
{ 9, 3 },
{ 9, 4 },
{ 10, 4 },
{ 11, 4 },
{ 12, 4 },
{ 12, 5 },
{ 13, 5 },
{ 14, 5 },
{ 15, 5 },
{ 15, 6 },
{ 16, 6 },
{ 16, 7 },
{ 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

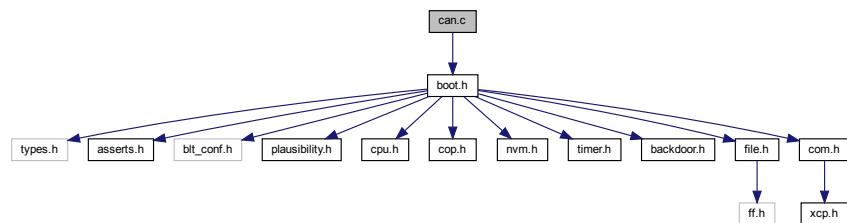
According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is $(\text{SYNC} + \text{TSEG1}) / (\text{SYNC} + \text{TSEG1} + \text{TSEG2}) * 100\%$. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

Referenced by [CanGetSpeedConfig\(\)](#).

7.20 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
Include dependency graph for HCS12/can.c:
```



Data Structures

- struct [tCanRxMsgSlot](#)
Structure type with the layout of a CAN reception message slot.
- struct [tCanTxMsgSlot](#)
Structure type with the layout of a CAN transmit message slot.
- struct [tCanRegs](#)
Structure type with the layout of the CAN related control registers.
- struct [tCanBusTiming](#)
Structure type for grouping CAN bus timing related information.

Macros

- `#define CAN_INIT_TIMEOUT_MS (250u)`
Timeout for entering/leaving CAN initialization mode in milliseconds.
- `#define CAN_MSG_TX_TIMEOUT_MS (50u)`
Timeout for transmitting a CAN message in milliseconds.
- `#define CAN_REGS_BASE_ADDRESS (0x0140)`
Set CAN base address to CAN0.
- `#define CAN ((volatile tCanRegs *)CAN_REGS_BASE_ADDRESS)`
Macro for accessing the CAN related control registers.
- `#define EXTIDMASK_BIT (0x80000000)`
Configures a CAN message id for 29-bit (extended).
- `#define INITRQ_BIT (0x01)`
Initialization mode request bit.
- `#define INITAK_BIT (0x01)`
Initialization mode handshake bit.
- `#define CANE_BIT (0x80)`
CAN controller enable bit.
- `#define IDAM0_BIT (0x10)`
Filter mode bit 0.
- `#define IDAM1_BIT (0x20)`
Filter mode bit 1.
- `#define TX0_BIT (0x01)`
Transmit buffer 0 select bit.
- `#define TXE0_BIT (0x01)`
Transmit buffer 0 empty bit.
- `#define IDE_BIT (0x08)`
29-bit extended id bit.
- `#define RXF_BIT (0x01)`
Receive buffer full flag bit.

Functions

- `static blt_bool CanGetSpeedConfig (blt_int16u baud, blt_int8u *btr0, blt_int8u *btr1)`
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- `void CanInit (void)`
Initializes the CAN controller and synchronizes it to the CAN bus.
- `void CanTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool CanReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.

Variables

- `static const tCanBusTiming canTiming []`
Array with possible time quanta configurations.

7.20.1 Detailed Description

Bootloader CAN communication interface source file.

7.20.2 Function Documentation

7.20.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u baud,
    blt_int8u * btr0,
    blt_int8u * btr1 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|-------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>btr0</i> | Pointer to where the value for register CANxBTR0 will be stored. |
| <i>btr1</i> | Pointer to where the value for register CANxBTR1 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.20.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.20.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.20.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.20.3 Variable Documentation**7.20.3.1 canTiming**

```
const tCanBusTiming canTiming[] [static]
```

Initial value:

```
=
{
    { 5, 2 },
    { 6, 2 },
    { 6, 3 },
    { 7, 3 },
    { 8, 3 },
    { 9, 3 },
    { 9, 4 },
    { 10, 4 },
    { 11, 4 },
    { 12, 4 },
    { 12, 5 },
    { 13, 5 },
    { 14, 5 },
    { 15, 5 },
```

```

{ 15, 6 },
{ 16, 6 },
{ 16, 7 },
{ 16, 8 }
}

```

Array with possible time quanta configurations.

According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2)

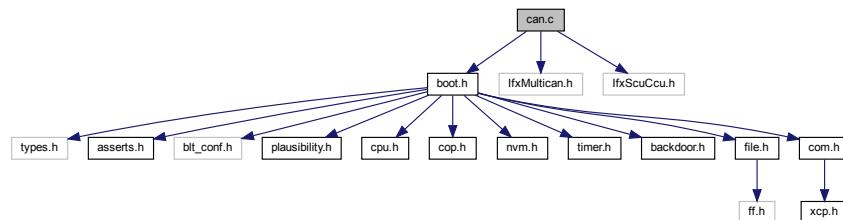
- 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

Referenced by [CanGetSpeedConfig\(\)](#).

7.21 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "IfxMultican.h"
#include "IfxScuCcu.h"
Include dependency graph for TRICORE_TC2/can.c:
```



Data Structures

- struct [tCanBusTiming](#)
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_INIT_TIMEOUT_MS** (250U)
Timeout for CAN module initialization operations in milliseconds.
- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.
- #define **CAN_MSG_TX_OBJECT_ID** ((IfxMultican_MsgObjId)0)
CAN node message object for transmitting messages.
- #define **CAN_MSG_RX_OBJECT_ID** ((IfxMultican_MsgObjId)1)
CAN node message object for receiving messages.

Functions

- static `blt_bool CanGetSpeedConfig (blt_int16u baud, blt_int16u *prescaler, blt_int8u *tseg1, blt_int8u *tseg2, blt_bool *div8)`
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void `CanInit (void)`
Initializes the CAN controller and synchronizes it to the CAN bus.
- void `CanTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool CanReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.

Variables

- `IfxMultican_Rxd_In * canRxPin = NULL_PTR`
Specifies the CAN Rx pin. It is expected that the application overwrites this value with the correct one, before `BootInit()` is called.
- `IfxMultican_Txd_Out * canTxPin = NULL_PTR`
Specifies the CAN Tx pin. It is expected that the application overwrites this value with the correct one, before `BootInit()` is called.
- static const `tCanBusTiming canTiming []`
CAN bittiming table for dynamically calculating the bittiming settings.

7.21.1 Detailed Description

Bootloader CAN communication interface source file.

7.21.2 Function Documentation

7.21.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u baud,
    blt_int16u * prescaler,
    blt_int8u * tseg1,
    blt_int8u * tseg2,
    blt_bool * div8 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------------|--|
| <code>baud</code> | The desired baudrate in kbps. Valid values are 10..1000. |
| <code>prescaler</code> | Pointer to where the value for the prescaler will be stored. |
| <code>tseg1</code> | Pointer to where the value for TSEG2 will be stored. |
| <code>tseg2</code> | Pointer to where the value for TSEG2 will be stored. |
| <code>div8</code> | BLT_TRUE if the extra divide-by-8 should be applied to the prescaler, BLT_FALSE otherwise. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

Referenced by [CanInit\(\)](#).

7.21.2.2 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.21.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.21.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.21.3 Variable Documentation

7.21.3.1 canTiming

```
const tCanBusTiming canTiming[] [static]
```

Initial value:

```
=  
{  
  
    { 5, 2 },  
    { 6, 2 },  
    { 6, 3 },  
    { 7, 3 },  
    { 8, 3 },  
    { 9, 3 },  
    { 9, 4 },  
    { 10, 4 },  
    { 11, 4 },  
    { 12, 4 },  
    { 12, 5 },  
    { 13, 5 },  
    { 14, 5 },  
    { 15, 5 },  
    { 15, 6 },  
    { 16, 6 },  
    { 16, 7 },  
    { 16, 8 }  
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

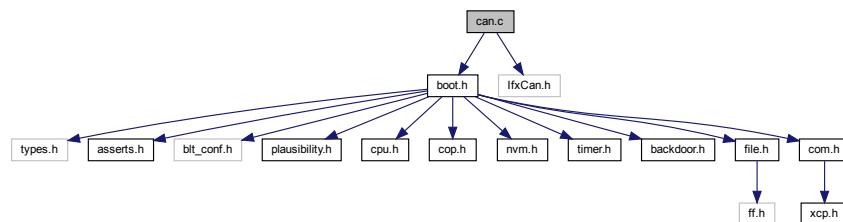
According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is $(\text{SYNC} + \text{TSEG1}) / (\text{SYNC} + \text{TSEG1} + \text{TSEG2}) * 100\%$. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

Referenced by [CanGetSpeedConfig\(\)](#).

7.22 can.c File Reference

Bootloader CAN communication interface source file.

```
#include "boot.h"
#include "IfxCAN.h"
Include dependency graph for TRICORE_TC3/can.c:
```



Data Structures

- struct [tCanBusTiming](#)
Structure type for grouping CAN bus timing related information.

Macros

- #define **CAN_MSG_TX_TIMEOUT_MS** (50u)
Timeout for transmitting a CAN message in milliseconds.
- #define **CAN_RX_FILTER_STD_MESSAGE_RAM_BASE_ADDR** (0x100)
Start address in message RAM for storing the message reception acceptance filters for 11-bit standard identifiers.
- #define **CAN_RX_FILTER_EXT_MESSAGE_RAM_BASE_ADDR** (0x200)
Start address in message RAM for storing the message reception acceptance filters for 29-bit extended identifiers.
- #define **CAN_RX_BUFFERS_MESSAGE_RAM_BASE_ADDR** (0x300)
Start address in message RAM for storing the received messages.
- #define **CAN_TX_BUFFERS_MESSAGE_RAM_BASE_ADDR** (0x400)
Start address in message RAM for storing the transmit messages.

Functions

- static [blt_bool CanGetSpeedConfig](#) ([blt_int16u](#) baud, [blt_int16u](#) *prescaler, [blt_int8u](#) *tseg1, [blt_int8u](#) *tseg2)
Search algorithm to match the desired baudrate to a possible bus timing configuration.
- void [CanInit](#) (void)
Initializes the CAN controller and synchronizes it to the CAN bus.
- void [CanTransmitPacket](#) ([blt_int8u](#) *data, [blt_int8u](#) len)
Transmits a packet formatted for the communication interface.
- [blt_bool CanReceivePacket](#) ([blt_int8u](#) *data, [blt_int8u](#) *len)
Receives a communication interface packet if one is present.

Variables

- IfxCAN_Rxd_In * **canRxPin** = NULL_PTR
Specifies the CAN Rx pin. It is expected that the application overwrites this value with the correct one, before [BootInit\(\)](#) is called.
- IfxCAN_Txd_Out * **canTxPin** = NULL_PTR
Specifies the CAN Tx pin. It is expected that the application overwrites this value with the correct one, before [BootInit\(\)](#) is called.
- static const **tCanBusTiming canTiming []**
CAN bittiming table for dynamically calculating the bittiming settings.
- static const IfxCAN_DataLengthCode **canDataLenLookup []**
Lookup table for converting the DLC value (0..8) into the data length code type that the driver library uses.

7.22.1 Detailed Description

Bootloader CAN communication interface source file.

7.22.2 Function Documentation

7.22.2.1 CanGetSpeedConfig()

```
static blt_bool CanGetSpeedConfig (
    blt_int16u baud,
    blt_int16u * prescaler,
    blt_int8u * tseg1,
    blt_int8u * tseg2 ) [static]
```

Search algorithm to match the desired baudrate to a possible bus timing configuration.

Parameters

| | |
|------------------|--|
| <i>baud</i> | The desired baudrate in kbps. Valid values are 10..1000. |
| <i>prescaler</i> | Pointer to where the value for the prescaler will be stored. |
| <i>tseg1</i> | Pointer to where the value for TSEG2 will be stored. |
| <i>tseg2</i> | Pointer to where the value for TSEG2 will be stored. |

Returns

BLT_TRUE if the CAN bustiming register values were found, BLT_FALSE otherwise.

7.22.2.2 CanInit()

```
void CanInit (
    void  )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

7.22.2.3 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE is a packet was received, BLT_FALSE otherwise.

7.22.2.4 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.22.3 Variable Documentation

7.22.3.1 canTiming

```
const tCanBusTiming canTiming[] [static]
```

Initial value:

```
=
{
    { 5, 2 },
    { 6, 2 },
    { 6, 3 },
    { 7, 3 },
    { 8, 3 },
    { 9, 3 },
    { 9, 4 },
    { 10, 4 },
    { 11, 4 },
    { 12, 4 },
    { 12, 5 },
    { 13, 5 },
    { 14, 5 },
    { 15, 5 },
    { 15, 6 },
    { 16, 6 },
    { 16, 7 },
    { 16, 8 }
}
```

CAN bittiming table for dynamically calculating the bittiming settings.

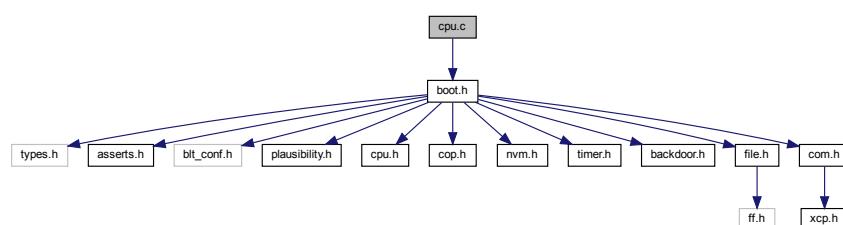
According to the CAN protocol 1 bit-time can be made up of between 8..25 time quanta (TQ). The total TQ in a bit is SYNC + TSEG1 + TSEG2 with SYNC always being 1. The sample point is (SYNC + TSEG1) / (SYNC + TSEG1 + SEG2) * 100%. This array contains possible and valid time quanta configurations with a sample point between 68..78%.

Referenced by [CanGetSpeedConfig\(\)](#).

7.23 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for _template/cpu.c:
```



Macros

- #define **CPU_USER_PROGRAM_VECTABLE_OFFSET** ((**blt_addr**)Nvm GetUserProgBaseAddress())
Pointer to the user program's vector table.

Functions

- void **CpuInit** (void)
Initializes the CPU module.
- void **CpuStartUserProgram** (void)
Starts the user program, if one is present. In this case this function does not return.
- void **CpuMemCopy** (**blt_addr** dest, **blt_addr** src, **blt_int16u** len)
Copies data from the source to the destination address.
- void **CpuMemSet** (**blt_addr** dest, **blt_int8u** value, **blt_int16u** len)
Sets the bytes at the destination address to the specified value.

7.23.1 Detailed Description

Bootloader cpu module source file.

7.23.2 Function Documentation

7.23.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

Referenced by [BootInit\(\)](#).

7.23.2.2 CpuMemCopy()

```
void CpuMemCopy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

Referenced by [FlashInitBlock\(\)](#), [XcpCmdGetSeed\(\)](#), [XcpCmdShortUpload\(\)](#), [XcpCmdUnlock\(\)](#), and [XcpCmdUpload\(\)](#).

7.23.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

Referenced by [XcpCmdShortUpload\(\)](#), and [XcpCmdUpload\(\)](#).

7.23.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

Returns

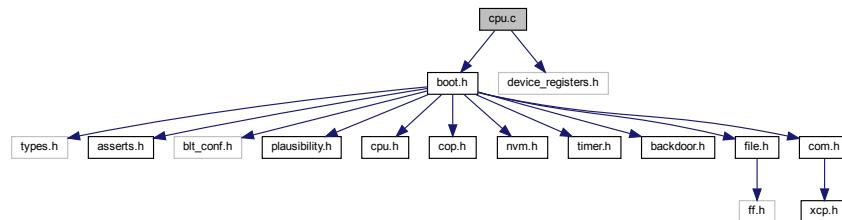
none.

Referenced by [BackDoorCheck\(\)](#), and [XcpCmdProgramReset\(\)](#).

7.24 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "device_registers.h"
Include dependency graph for ARMCM0_S32K11/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_addr)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- void `CpuInit` (void)
Initializes the CPU module.
- void `CpuStartUserProgram` (void)
Starts the user program, if one is present. In this case this function does not return.
- void `CpuMemcpy` (blt_addr dest, blt_addr src, blt_int16u len)
Copies data from the source to the destination address.
- void `CpuMemSet` (blt_addr dest, blt_int8u value, blt_int16u len)
Sets the bytes at the destination address to the specified value.

7.24.1 Detailed Description

Bootloader cpu module source file.

7.24.2 Function Documentation

7.24.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.24.2.2 CpuMemCopy()

```
void CpuMemCopy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.24.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.24.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

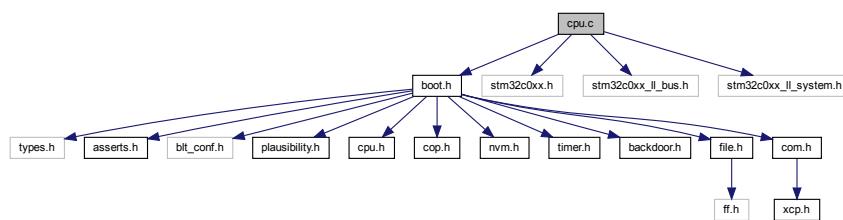
Returns

none.

7.25 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32c0xx.h"
#include "stm32c0xx_ll_bus.h"
#include "stm32c0xx_ll_system.h"
Include dependency graph for ARMCM0_STM32C0/cpu.c:
```

**Macros**

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- void **CpuInit** (void)
Initializes the CPU module.
- void **CpuStartUserProgram** (void)
Starts the user program, if one is present. In this case this function does not return.
- void **CpuMemCopy** (**blt_addr** dest, **blt_addr** src, **blt_int16u** len)
Copies data from the source to the destination address.
- void **CpuMemSet** (**blt_addr** dest, **blt_int8u** value, **blt_int16u** len)
Sets the bytes at the destination address to the specified value.

7.25.1 Detailed Description

Bootloader cpu module source file.

7.25.2 Function Documentation

7.25.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.25.2.2 CpuMemCopy()

```
void CpuMemCopy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.25.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.25.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

Returns

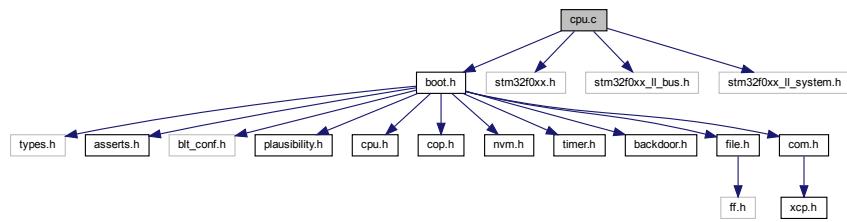
none.

7.26 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32f0xx.h"
#include "stm32f0xx_ll_bus.h"
```

```
#include "stm32f0xx_ll_system.h"
Include dependency graph for ARMCM0_STM32F0/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.
- `#define CPU_USER_PROGRAM_VECTABLE_SIZE (0xC0u)`
Size in bytes of the user program's vector table.
- `#define CPU_USER_PROGRAM_RAM_BASEADDR ((blt_addr)(0x20000000))`
Start address of the user program's RAM where its vector table will be copied to.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.26.1 Detailed Description

Bootloader cpu module source file.

7.26.2 Function Documentation

7.26.2.1 CpuInit()

```
void CpuInit (
    void  )
```

Initializes the CPU module.

Returns

none.

7.26.2.2 CpuMemCopy()

```
void CpuMemCopy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.26.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.26.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

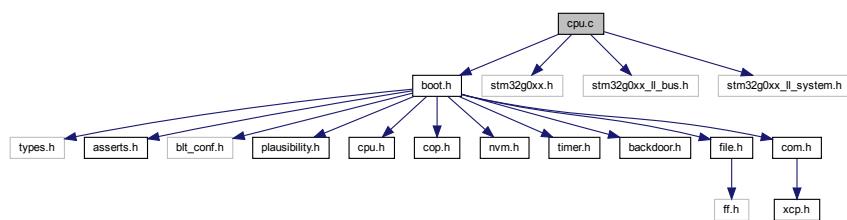
Returns

none.

7.27 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32g0xx.h"
#include "stm32g0xx_ll_bus.h"
#include "stm32g0xx_ll_system.h"
Include dependency graph for ARCMC0_STM32G0/cpu.c:
```

**Macros**

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- void **CpuInit** (void)

Initializes the CPU module.
- void **CpuStartUserProgram** (void)

Starts the user program, if one is present. In this case this function does not return.
- void **CpuMemCopy** (**blt_addr** dest, **blt_addr** src, **blt_int16u** len)

Copies data from the source to the destination address.
- void **CpuMemSet** (**blt_addr** dest, **blt_int8u** value, **blt_int16u** len)

Sets the bytes at the destination address to the specified value.

7.27.1 Detailed Description

Bootloader cpu module source file.

7.27.2 Function Documentation

7.27.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.27.2.2 CpuMemCopy()

```
void CpuMemCopy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.27.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.27.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

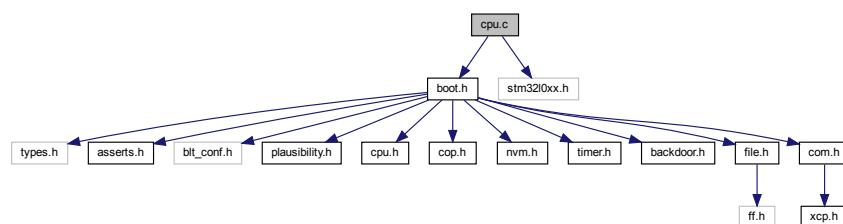
Returns

none.

7.28 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm3210xx.h"
Include dependency graph for ARMCM0_STM32L0/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.28.1 Detailed Description

Bootloader cpu module source file.

7.28.2 Function Documentation

7.28.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.28.2.2 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.28.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.28.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

Returns

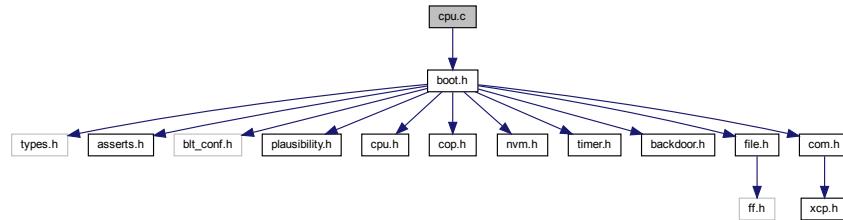
none.

7.29 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM0_XMC1/cpu.c:



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.29.1 Detailed Description

Bootloader cpu module source file.

7.29.2 Function Documentation

7.29.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.29.2.2 CpuMemCopy()

```
void CpuMemCopy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.29.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.29.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

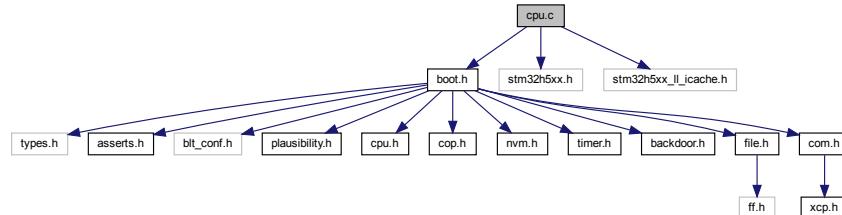
Returns

none.

7.30 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32h5xx.h"
#include "stm32h5xx_ll_icache.h"
Include dependency graph for ARMCM33_STM32H5/cpu.c:
```

**Macros**

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- void **CpuInit** (void)
Initializes the CPU module.
- void **CpuStartUserProgram** (void)
Starts the user program, if one is present. In this case this function does not return.
- void **CpuMemCopy** (**blt_addr** dest, **blt_addr** src, **blt_int16u** len)
Copies data from the source to the destination address.
- void **CpuMemSet** (**blt_addr** dest, **blt_int8u** value, **blt_int16u** len)
Sets the bytes at the destination address to the specified value.

7.30.1 Detailed Description

Bootloader cpu module source file.

7.30.2 Function Documentation

7.30.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.30.2.2 CpuMemCopy()

```
void CpuMemCopy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.30.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.30.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

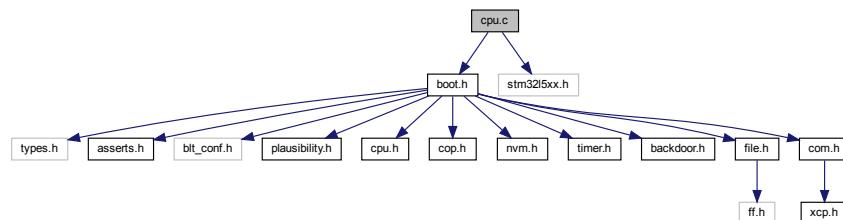
Returns

none.

7.31 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm3215xx.h"
Include dependency graph for ARMCM33_STM32L5/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.31.1 Detailed Description

Bootloader cpu module source file.

7.31.2 Function Documentation

7.31.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.31.2.2 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.31.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.31.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

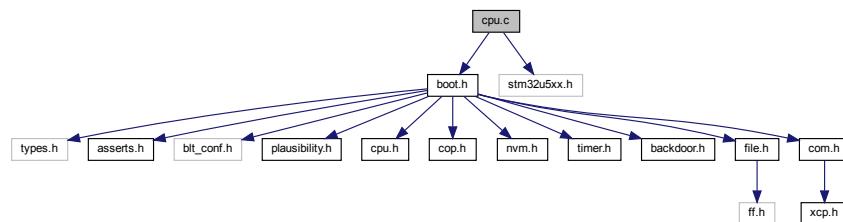
Returns

none.

7.32 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32u5xx.h"
Include dependency graph for ARMCM33_STM32U5/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.32.1 Detailed Description

Bootloader cpu module source file.

7.32.2 Function Documentation

7.32.2.1 CpuInit()

```
void CpuInit (
    void  )
```

Initializes the CPU module.

Returns

none.

7.32.2.2 CpuMemCopy()

```
void CpuMemCopy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.32.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.32.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

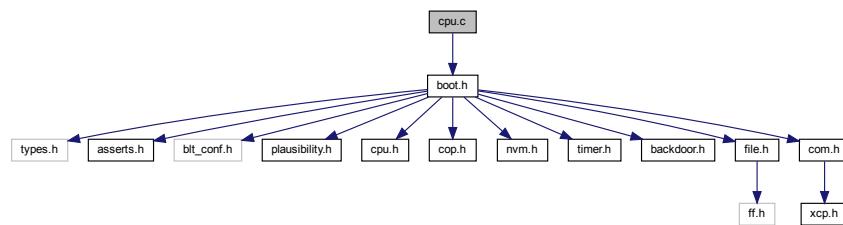
Returns

none.

7.33 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_EFM32/cpu.c:
```

**Macros**

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.
- `#define SCB_VTOR (*((volatile blt_int32u *) 0xE000ED08))`
Vector table offset register.

Functions

- void **CpuInit** (void)

Initializes the CPU module.
- void **CpuStartUserProgram** (void)

Starts the user program, if one is present. In this case this function does not return.
- void **CpuMemCopy** (**blt_addr** dest, **blt_addr** src, **blt_int16u** len)

Copies data from the source to the destination address.
- void **CpuMemSet** (**blt_addr** dest, **blt_int8u** value, **blt_int16u** len)

Sets the bytes at the destination address to the specified value.

7.33.1 Detailed Description

Bootloader cpu module source file.

7.33.2 Function Documentation

7.33.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.33.2.2 CpuMemCopy()

```
void CpuMemCopy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.33.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.33.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

Returns

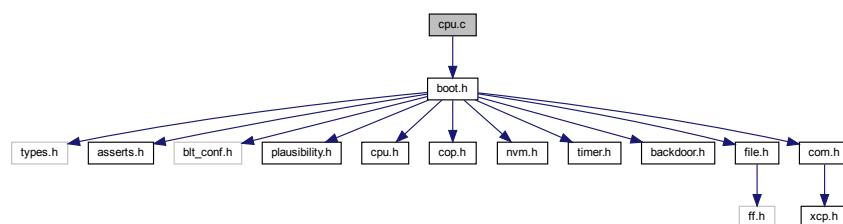
none.

7.34 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM3_LM3S/cpu.c:



Macros

- #define **CPU_USER_PROGRAM_STARTADDR_PTR** ((**blt_addr**)(Nvm GetUserProgBaseAddress() + 0x00000004))
Pointer to the user program's reset vector.
- #define **CPU_USER_PROGRAM_VECTABLE_OFFSET** ((**blt_int32u**)Nvm GetUserProgBaseAddress())
Pointer to the user program's vector table.
- #define **SCB_VTOR** (*((volatile **blt_int32u** *) 0xE000ED08))
Vector table offset register.

Functions

- void **CpuInit** (void)
Initializes the CPU module.
- void **CpuStartUserProgram** (void)
Starts the user program, if one is present. In this case this function does not return.
- void **CpuMemcpy** (**blt_addr** dest, **blt_addr** src, **blt_int16u** len)
Copies data from the source to the destination address.
- void **CpuMemSet** (**blt_addr** dest, **blt_int8u** value, **blt_int16u** len)
Sets the bytes at the destination address to the specified value.

7.34.1 Detailed Description

Bootloader cpu module source file.

7.34.2 Function Documentation

7.34.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.34.2.2 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.34.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.34.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

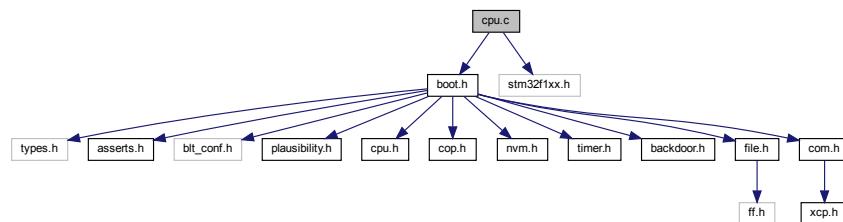
Returns

none.

7.35 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32f1xx.h"
Include dependency graph for ARMCM3_STM32F1/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- void `CpuInit` (void)
Initializes the CPU module.
- void `CpuStartUserProgram` (void)
Starts the user program, if one is present. In this case this function does not return.
- void `CpuMemcpy` (blt_addr dest, blt_addr src, blt_int16u len)
Copies data from the source to the destination address.
- void `CpuMemSet` (blt_addr dest, blt_int8u value, blt_int16u len)
Sets the bytes at the destination address to the specified value.

7.35.1 Detailed Description

Bootloader cpu module source file.

7.35.2 Function Documentation

7.35.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.35.2.2 CpuMemCopy()

```
void CpuMemCopy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.35.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.35.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

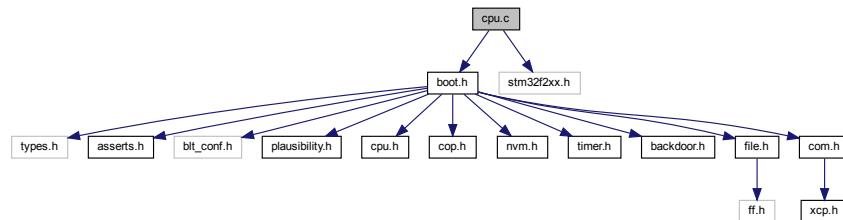
Returns

none.

7.36 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32f2xx.h"
Include dependency graph for ARMCM3_STM32F2/cpu.c:
```

**Macros**

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.36.1 Detailed Description

Bootloader cpu module source file.

7.36.2 Function Documentation

7.36.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.36.2.2 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.36.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
```

```
blt_int8u value,
blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.36.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

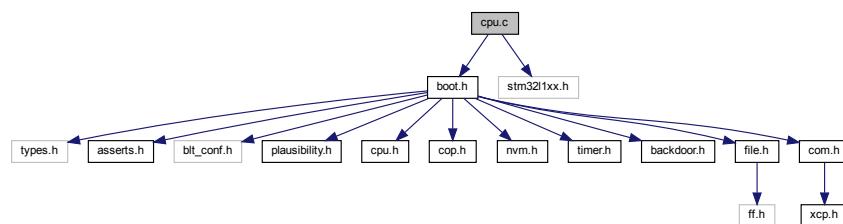
Returns

none.

7.37 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32l1xx.h"
Include dependency graph for ARMCM3_STM32L1/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.37.1 Detailed Description

Bootloader cpu module source file.

7.37.2 Function Documentation

7.37.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.37.2.2 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.37.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.37.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

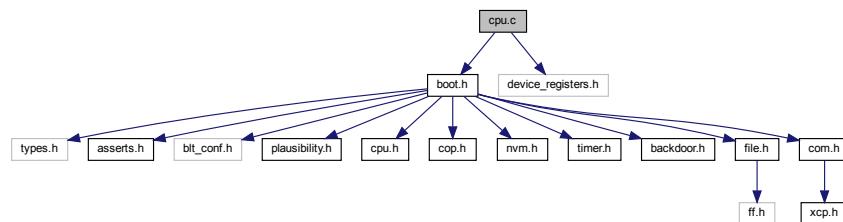
Returns

none.

7.38 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "device_registers.h"
Include dependency graph for ARMCM4_S32K14/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_addr)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- void `CpuInit` (void)
Initializes the CPU module.
- void `CpuStartUserProgram` (void)
Starts the user program, if one is present. In this case this function does not return.
- void `CpuMemcpy` (blt_addr dest, blt_addr src, blt_int16u len)
Copies data from the source to the destination address.
- void `CpuMemSet` (blt_addr dest, blt_int8u value, blt_int16u len)
Sets the bytes at the destination address to the specified value.

7.38.1 Detailed Description

Bootloader cpu module source file.

7.38.2 Function Documentation

7.38.2.1 CpuInit()

```
void CpuInit (
    void  )
```

Initializes the CPU module.

Returns

none.

7.38.2.2 CpuMemCopy()

```
void CpuMemCopy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.38.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.38.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

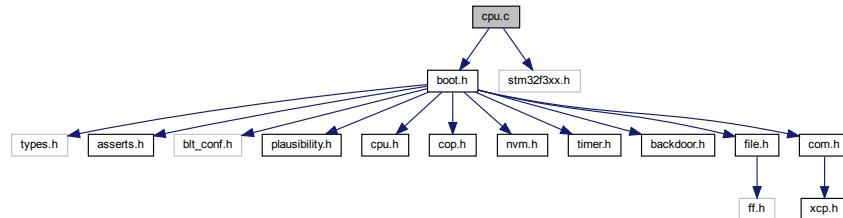
Returns

none.

7.39 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32f3xx.h"
Include dependency graph for ARMCM4_STM32F3/cpu.c:
```

**Macros**

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.39.1 Detailed Description

Bootloader cpu module source file.

7.39.2 Function Documentation

7.39.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.39.2.2 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.39.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
```

```
blt_int8u value,
blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.39.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

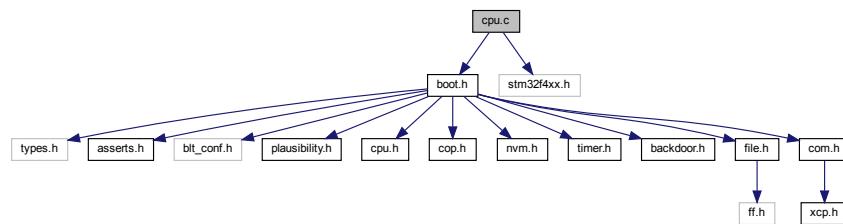
Returns

none.

7.40 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32f4xx.h"
Include dependency graph for ARMCM4_STM32F4/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.40.1 Detailed Description

Bootloader cpu module source file.

7.40.2 Function Documentation

7.40.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.40.2.2 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.40.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.40.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

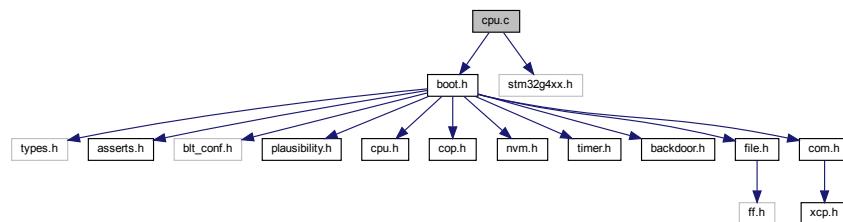
Returns

none.

7.41 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32g4xx.h"
Include dependency graph for ARMCM4_STM32G4/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_addr)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- void `CpuInit` (void)
Initializes the CPU module.
- void `CpuStartUserProgram` (void)
Starts the user program, if one is present. In this case this function does not return.
- void `CpuMemcpy` (blt_addr dest, blt_addr src, blt_int16u len)
Copies data from the source to the destination address.
- void `CpuMemSet` (blt_addr dest, blt_int8u value, blt_int16u len)
Sets the bytes at the destination address to the specified value.

7.41.1 Detailed Description

Bootloader cpu module source file.

7.41.2 Function Documentation

7.41.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.41.2.2 CpuMemCopy()

```
void CpuMemCopy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.41.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.41.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

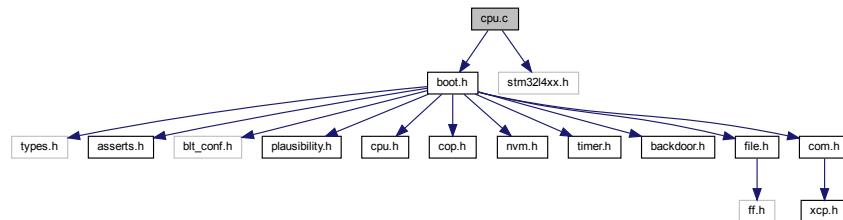
Returns

none.

7.42 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32l4xx.h"
Include dependency graph for ARMCM4_STM32L4/cpu.c:
```

**Macros**

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.42.1 Detailed Description

Bootloader cpu module source file.

7.42.2 Function Documentation

7.42.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.42.2.2 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.42.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
```

```
blt_int8u value,
blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.42.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

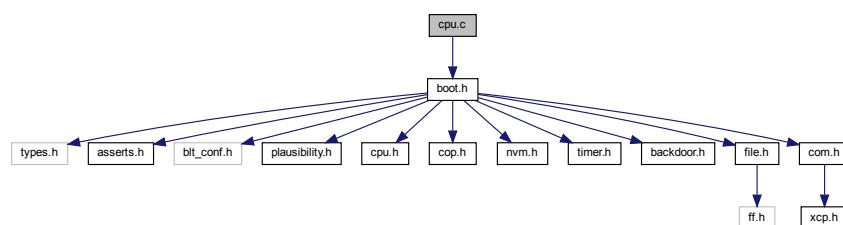
Returns

none.

7.43 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_TM4C/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.
- `#define SCB_VTOR (*((volatile blt_int32u *) 0xE000ED08))`
Vector table offset register.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.43.1 Detailed Description

Bootloader cpu module source file.

7.43.2 Function Documentation

7.43.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.43.2.2 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.43.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.43.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

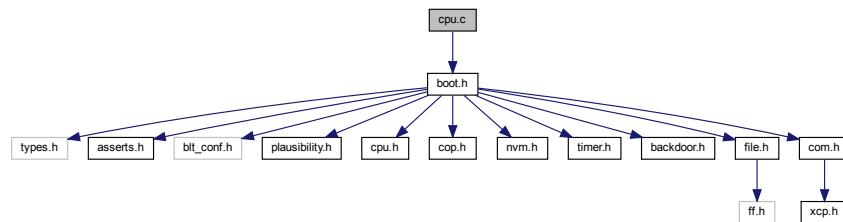
Returns

none.

7.44 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_XMC4/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.
- `#define SCB_VTOR (*((volatile blt_int32u *) 0xE000ED08))`
Vector table offset register.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.44.1 Detailed Description

Bootloader cpu module source file.

7.44.2 Function Documentation

7.44.2.1 CpuInit()

```
void CpuInit (
    void  )
```

Initializes the CPU module.

Returns

none.

7.44.2.2 CpuMemCopy()

```
void CpuMemCopy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.44.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.44.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

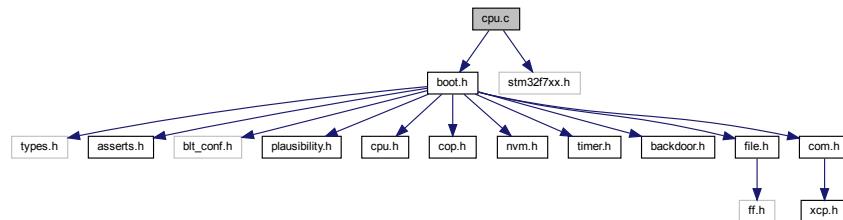
Returns

none.

7.45 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32f7xx.h"
Include dependency graph for ARMCM7_STM32F7/cpu.c:
```

**Macros**

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_int32u)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.45.1 Detailed Description

Bootloader cpu module source file.

7.45.2 Function Documentation

7.45.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.45.2.2 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.45.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
```

```
blt_int8u value,
blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.45.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

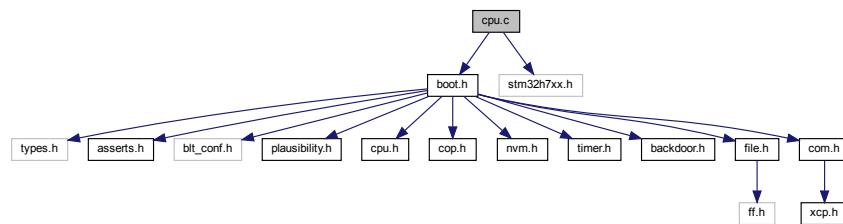
Returns

none.

7.46 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "stm32h7xx.h"
Include dependency graph for ARMCM7_STM32H7/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x00000004))`
Pointer to the user program's reset vector.
- `#define CPU_USER_PROGRAM_VECTABLE_OFFSET ((blt_addr)Nvm GetUserProgBaseAddress())`
Pointer to the user program's vector table.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.46.1 Detailed Description

Bootloader cpu module source file.

7.46.2 Function Documentation

7.46.2.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.46.2.2 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.46.2.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.46.2.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

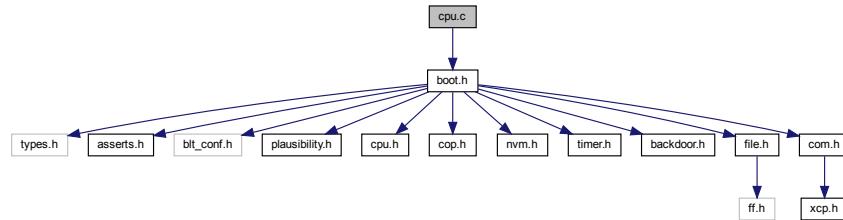
Returns

none.

7.47 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for HCS12/cpu.c:
```



Macros

- `#define CPU_USER_PROGRAM_STARTADDR_PTR (Nvm GetUserProgBaseAddress() - 2)`
Start address of the user program. This is the address of the reset vector in the user program's vector table.

Functions

- `void CpuInit (void)`
Initializes the CPU module.
- `void CpuStartUserProgram (void)`
Starts the user program, if one is present. In this case this function does not return.
- `void CpuMemcpy (blt_addr dest, blt_addr src, blt_int16u len)`
Copies data from the source to the destination address.
- `void CpuMemSet (blt_addr dest, blt_int8u value, blt_int16u len)`
Sets the bytes at the destination address to the specified value.

7.47.1 Detailed Description

Bootloader cpu module source file.

7.47.2 Macro Definition Documentation

7.47.2.1 CPU_USER_PROGRAM_STARTADDR_PTR

```
#define CPU_USER_PROGRAM_STARTADDR_PTR (Nvm GetUserProgBaseAddress() - 2)
```

Start address of the user program. This is the address of the reset vector in the user program's vector table.

Attention

This value must be updated if the memory reserved for the bootloader changes.

7.47.3 Function Documentation

7.47.3.1 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.47.3.2 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.47.3.3 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.47.3.4 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

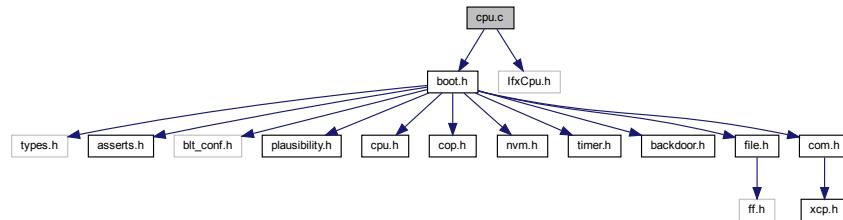
Returns

none.

7.48 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "IfxCpu.h"
Include dependency graph for TRICORE_TC2/cpu.c:
```



Macros

- #define **CPU_USER_PROGRAM_STARTADDR_PTR** ((blt_addr)(Nvm GetUserProgBaseAddress() + 0x000000020))
Pointer to the user program's reset vector.

Functions

- static void **CpuEnableUncorrectableBitErrorTrap** (blt_bool enable)
Enables or disables the reporting of an uncorrectable bit error to the CPU. On this microcontroller, directly reading data from flash memory can result in a CPU trap, when that particular flash page was not previously newly programmed (IfxCpu_Trap_Bus_Id_dataAccessSynchronousError).
- void **CpuInit** (void)
Initializes the CPU module.
- void **CpuStartUserProgram** (void)
Starts the user program, if one is present. In this case this function does not return.
- void **CpuMemcpy** (blt_addr dest, blt_addr src, blt_int16u len)
Copies data from the source to the destination address.
- void **CpuMemSet** (blt_addr dest, blt_int8u value, blt_int16u len)
Sets the bytes at the destination address to the specified value.

7.48.1 Detailed Description

Bootloader cpu module source file.

7.48.2 Function Documentation

7.48.2.1 CpuEnableUncorrectableBitErrorTrap()

```
static void CpuEnableUncorrectableBitErrorTrap (
    blt_bool enable ) [static]
```

Enables or disables the reporting of an uncorrectable bit error to the CPU. On this microcontroller, directly reading data from flash memory can result in a CPU trap, when that particular flash page was not previously newly programmed (IfxCpu_Trap_Bus_Id_dataAccessSynchronousError).

Parameters

| | |
|---------------------|--|
| <code>enable</code> | BLT_TRUE to enable generation of the CPU trap, BLT_FALSE to disable. |
|---------------------|--|

Returns

none.

Referenced by [CpuInit\(\)](#).

7.48.2.2 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.48.2.3 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.48.2.4 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.48.2.5 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

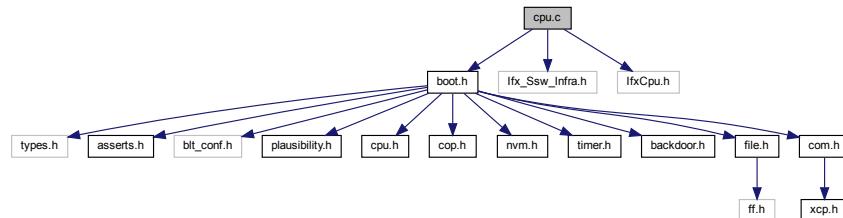
Returns

none.

7.49 cpu.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "Ifx_Ssw_Infra.h"
#include "IfxCpu.h"
Include dependency graph for TRICORE_TC3/cpu.c:
```

**Functions**

- static void [CpuEnableUncorrectableEccErrorTrap](#) (*blt_bool* enable)

Enables or disables the reporting of an uncorrectable ECC error to the CPU. On this microcontroller, directly reading data from flash memory can result in a CPU trap, when that particular flash page is in the erased state. This is caused because an unprogrammed flash page also doesn't have its ECC bits set.
- void [CpuInit](#) (*void*)

Initializes the CPU module.
- void [CpuStartUserProgram](#) (*void*)

Starts the user program, if one is present. In this case this function does not return.
- void [CpuMemcpy](#) (*blt_addr* dest, *blt_addr* src, *blt_int16u* len)

Copies data from the source to the destination address.
- void [CpuMemSet](#) (*blt_addr* dest, *blt_int8u* value, *blt_int16u* len)

Sets the bytes at the destination address to the specified value.

7.49.1 Detailed Description

Bootloader cpu module source file.

7.49.2 Function Documentation

7.49.2.1 CpuEnableUncorrectableEccErrorTrap()

```
static void CpuEnableUncorrectableEccErrorTrap (
    blt_bool enable ) [static]
```

Enables or disables the reporting of an uncorrectable ECC error to the CPU. On this microcontroller, directly reading data from flash memory can result in a CPU trap, when that particular flash page is in the erased state. This is caused because an unprogrammed flash page also doesn't have its ECC bits set.

Parameters

| | |
|---------------|--|
| <i>enable</i> | BLT_TRUE to enable generation of the CPU trap, BLT_FALSE to disable. |
|---------------|--|

Returns

none.

Referenced by [CpuInit\(\)](#).

7.49.2.2 CpuInit()

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

7.49.2.3 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

7.49.2.4 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

7.49.2.5 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

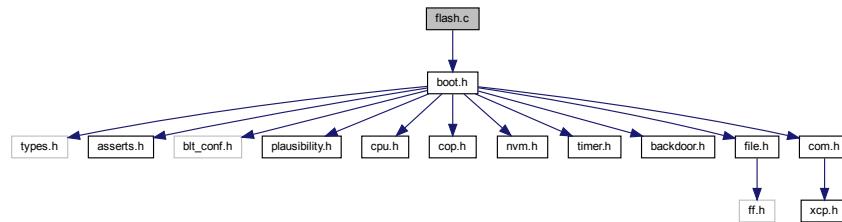
Returns

none.

7.50 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
Include dependency graph for _template/flash.c:
```



Data Structures

- struct [tFlashSector](#)
Flash sector descriptor type.
- struct [tFlashBlockInfo](#)
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR_IDX** (0xff)
Value for an invalid sector entry index into flashLayout[].
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SECTORS** (sizeof(flashLayout)/sizeof(flashLayout[0]))
Total numbers of sectors in array flashLayout[].
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x188)
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- static `blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- static `tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- static `blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static `blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static `blt_bool FlashEraseSectors (blt_int8u first_sector_idx, blt_int8u last_sector_idx)`
Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.
- static `blt_int8u FlashGetSectorIdx (blt_addr address)`
Determines the index into the flashLayout[] array of the flash sector that the specified address is in.
- void `FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- static `tFlashBlockInfo blockInfo`
Local variable with information about the flash block that is currently being operated on.
- static `tFlashBlockInfo bootBlockInfo`
Local variable with information about the flash boot block.

7.50.1 Detailed Description

Bootloader flash driver source file.

7.50.2 Function Documentation

7.50.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.50.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.50.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.50.2.4 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector_idx,
    blt_int8u last_sector_idx ) [static]
```

Erases the flash sectors from indices *first_sector_idx* up until *last_sector_idx* into the *flashLayout[]* array.

Parameters

| | |
|-------------------------|---|
| <i>first_sector_idx</i> | First flash sector number index into <i>flashLayout[]</i> . |
| <i>last_sector_idx</i> | Last flash sector number index into <i>flashLayout[]</i> . |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.50.2.5 FlashGetSectorIdx()

```
static blt_int8u FlashGetSectorIdx (
    blt_addr address ) [static]
```

Determines the index into the *flashLayout[]* array of the flash sector that the specified address is in.

Parameters

| | |
|----------------|------------------------------|
| <i>address</i> | Address in the flash sector. |
|----------------|------------------------------|

Returns

Flash sector index in *flashLayout[]* or FLASH_INVALID_SECTOR_IDX.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.50.2.6 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Referenced by [Nvm GetUserProgBaseAddress\(\)](#).

7.50.2.7 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.50.2.8 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.50.2.9 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is now being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.50.2.10 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [NvmVerifyChecksum\(\)](#).

7.50.2.11 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.50.2.12 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.50.2.13 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.50.3 Variable Documentation

7.50.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.50.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.50.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

Initial value:

```
=
{

    { 0x08008000, 0x4000, 2 },
    { 0x0800C000, 0x4000, 3 },
    { 0x08010000, 0x4000, 4 },
    { 0x08014000, 0x4000, 5 },
    { 0x08018000, 0x4000, 6 },
    { 0x0801C000, 0x4000, 7 },
    { 0x08020000, 0x4000, 8 },
    { 0x08024000, 0x4000, 9 },
    { 0x08028000, 0x4000, 10 },
    { 0x0802C000, 0x4000, 11 },
    { 0x08030000, 0x4000, 12 },
    { 0x08034000, 0x4000, 13 },
    { 0x08038000, 0x4000, 14 },
    { 0x0803C000, 0x4000, 15 },
```

}

If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to > 0 in `blt_conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.

Array wit the layout of the flash memory.

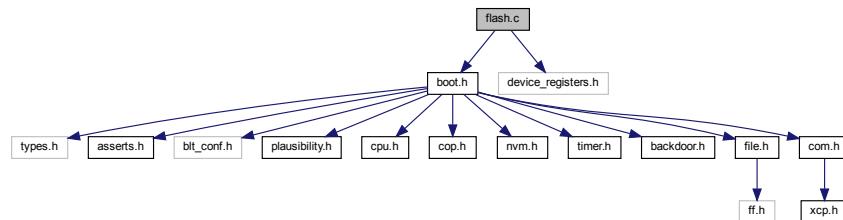
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

Referenced by [FlashEraseSectors\(\)](#), [FlashGetSectorIdx\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.51 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "device_registers.h"
Include dependency graph for ARMCM0_S32K11/flash.c:
```



Data Structures

- struct [tFlashSector](#)
Flash sector descriptor type.
- struct [tFlashBlockInfo](#)
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR_IDX** (0xff)
Value for an invalid sector entry index into flashLayout[].
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (256)
Standard size of a flash block for writing.
- #define **FLASH_ERASE_BLOCK_SIZE** (FEATURE_FLS_PF_BLOCK_SECTOR_SIZE)
Standard size of a flash block for erasing. This is either 2 or 4 kb depending on the microcontroller derivative.

- `#define FLASH_TOTAL_SECTORS (sizeof(flashLayout)/sizeof(flashLayout[0]))`
Total numbers of sectors in array flashLayout[].
- `#define FLASH_END_ADDRESS`
End address of the bootloader programmable flash.
- `#define FLASH_FTFC_CMD_PROGRAM_PHRASE (0x07U)`
FTFC program phrase command code.
- `#define FLASH_FTFC_CMD_ERASE_SECTOR (0x09U)`
FTFC erase sector command code.
- `#define BOOT_FLASH_VECTOR_TABLE_CS_OFFSET (0xC0)`
Offset into the user program's vector table where the checksum is located. Note that the value can be overriden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- `static blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- `static tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- `static blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- `static blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- `static blt_bool FlashEraseSectors (blt_int8u first_sector_idx, blt_int8u last_sector_idx)`
Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.
- `static blt_int8u FlashGetSectorIdx (blt_addr address)`
Determines the index into the flashLayout[] array of the flash sector that the specified address is in.
- `static START_FUNCTION_DECLARATION_RAMSECTION void FlashCommandSequence (void)`
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static [tFlashBlockInfo blockInfo](#)
Local variable with information about the flash block that is currently being operated on.
- static [tFlashBlockInfo bootBlockInfo](#)
Local variable with information about the flash boot block.

7.51.1 Detailed Description

Bootloader flash driver source file.

7.51.2 Function Documentation

7.51.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.51.2.2 FlashCommandSequence()

```
static START_FUNCTION_DEFINITION_RAMSECTION void FlashCommandSequence (
    void ) [static]
```

If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to `> 0` in `blt_conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.

Use the FTFC module to run the flash command sequence. It is assumed that that command and its necessary parameters were already written to the correct FTFC registers.

Array wit the layout of the flash memory.

Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

Attention

This function needs to run from RAM. It is configured such that the C start-up code automatically copies it from ROM to RAM in function `init_data_bss()`, which is called by the reset handler.

Returns

None.

Referenced by [FlashEraseSectors\(\)](#), and [FlashWriteBlock\(\)](#).

7.51.2.3 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

`BLT_TRUE` if successful, `BLT_FALSE` otherwise.

7.51.2.4 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------------|------------------|
| <code>addr</code> | Start address. |
| <code>len</code> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.51.2.5 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector_idx,
    blt_int8u last_sector_idx ) [static]
```

Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.

Parameters

| | |
|-------------------------|---|
| <i>first_sector_idx</i> | First flash sector number index into flashLayout[]. |
| <i>last_sector_idx</i> | Last flash sector number index into flashLayout[]. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.51.2.6 FlashGetSectorIdx()

```
static blt_int8u FlashGetSectorIdx (
    blt_addr address ) [static]
```

Determines the index into the flashLayout[] array of the flash sector that the specified address is in.

Parameters

| | |
|----------------|------------------------------|
| <i>address</i> | Address in the flash sector. |
|----------------|------------------------------|

Returns

Flash sector index in flashLayout[] or FLASH_INVALID_SECTOR_IDX.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.51.2.7 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.51.2.8 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.51.2.9 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.51.2.10 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is now being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.51.2.11 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.51.2.12 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.51.2.13 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.51.2.14 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.51.3 Variable Documentation

7.51.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.51.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

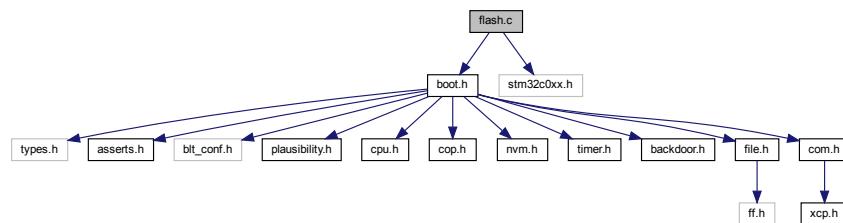
The first block of the user program holds the vector table, which on the STM32 is also the where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.52 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32c0xx.h"
Include dependency graph for ARMCM0_STM32C0/flash.c:
```



Data Structures

- struct [tFlashSector](#)
Flash sector descriptor type.
- struct [tFlashBlockInfo](#)
Structure type for grouping flash block information.

Macros

- `#define FLASH_INVALID_SECTOR (0xff)`
Value for an invalid sector number.
- `#define FLASH_INVALID_ADDRESS (0xffffffff)`
Value for an invalid flash address.
- `#define FLASH_WRITE_BLOCK_SIZE (512)`
Standard size of a flash block for writing.
- `#define FLASH_TOTAL_SECTORS (sizeof(FLASH_LAYOUT)/sizeof(FLASH_LAYOUT[0]))`
Total numbers of sectors in array `FLASH_LAYOUT[]`.
- `#define FLASH_END_ADDRESS`
End address of the bootloader programmable flash.
- `#define FLASH_ERASE_BLOCK_SIZE (FLASH_PAGE_SIZE)`
Hardware erase unit size (sectors must be multiples of this)
- `#define BOOT_FLASH_VECTOR_TABLE_CS_OFFSET (0xC0)`
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in `blt_conf.h`, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- `static blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- `static tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- `static blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- `static blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- `static blt_bool FlashEraseSectors (blt_int8u first_sector, blt_int8u last_sector)`
Erases the flash sectors from first_sector up until last_sector.
- `static blt_int8u FlashGetSector (blt_addr address)`
Determines the flash sector the address is in.
- `static blt_addr FlashGetSectorBaseAddr (blt_int8u sector)`
Determines the flash sector base address.
- `static blt_addr FlashGetSectorSize (blt_int8u sector)`
Determines the flash sector size.
- `static blt_int32u FlashGetPage (blt_addr address)`
Determines the flash page that the address belongs to.
- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

- **blt_bool FlashVerifyChecksum (void)**
Verifies the checksum, which indicates that a valid user program is present and can be started.
- **blt_bool FlashDone (void)**
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- **blt_addr Flash GetUserProgBaseAddress (void)**
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const **tFlashSector flashLayout []**
If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to > 0 in `blt_conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.
- static **tFlashBlockInfo blockInfo**
Local variable with information about the flash block that is currently being operated on.
- static **tFlashBlockInfo bootBlockInfo**
Local variable with information about the flash boot block.

7.52.1 Detailed Description

Bootloader flash driver source file.

7.52.2 Function Documentation

7.52.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.52.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.52.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.52.2.4 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector,
    blt_int8u last_sector ) [static]
```

Erases the flash sectors from *first_sector* up until *last_sector*.

Parameters

| | |
|---------------------|----------------------------|
| <i>first_sector</i> | First flash sector number. |
| <i>last_sector</i> | Last flash sector number. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.52.2.5 FlashGetPage()

```
static blt_int32u FlashGetPage (
    blt_addr address ) [static]
```

Determines the flash page that the address belongs to.

Parameters

| | |
|----------------|-----------------------|
| <i>address</i> | Flash memory address. |
|----------------|-----------------------|

Returns

Page number.

Referenced by [FlashEraseSectors\(\)](#).

7.52.2.6 FlashGetSector()

```
static blt_int8u FlashGetSector (
    blt_addr address ) [static]
```

Determines the flash sector the address is in.

Parameters

| | |
|----------------|------------------------------|
| <i>address</i> | Address in the flash sector. |
|----------------|------------------------------|

Returns

Flash sector number or FLASH_INVALID_SECTOR.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.52.2.7 FlashGetSectorBaseAddr()

```
static blt_addr FlashGetSectorBaseAddr (
    blt_int8u sector ) [static]
```

Determines the flash sector base address.

Parameters

| | |
|---------------|------------------------------------|
| <i>sector</i> | Sector to get the base address of. |
|---------------|------------------------------------|

Returns

Flash sector base address or FLASH_INVALID_ADDRESS.

Referenced by [FlashEraseSectors\(\)](#).

7.52.2.8 FlashGetSectorSize()

```
static blt_addr FlashGetSectorSize (
    blt_int8u sector ) [static]
```

Determines the flash sector size.

Parameters

| | |
|---------------|----------------------------|
| <i>sector</i> | Sector to get the size of. |
|---------------|----------------------------|

Returns

Flash sector size or 0.

Referenced by [FlashEraseSectors\(\)](#).

7.52.2.9 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.52.2.10 FlashInit()

```
void FlashInit (
    void  )
```

Initializes the flash driver.

Returns

none.

7.52.2.11 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.52.2.12 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is now being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.52.2.13 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.52.2.14 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.52.2.15 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.52.2.16 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.52.3 Variable Documentation

7.52.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.52.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also the where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.52.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory.

Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten. The current flash layout does not reflect the minimum sector size of the physical flash (2 KiB), because this would make the table quite long and a waste of ROM. The minimum sector size is only really needed when erasing the flash. This can still be done in combination with macro FLASH_ERASE_BLOCK_SIZE. Note that the term sector here is used in a different meaning than in the controller's reference manual.

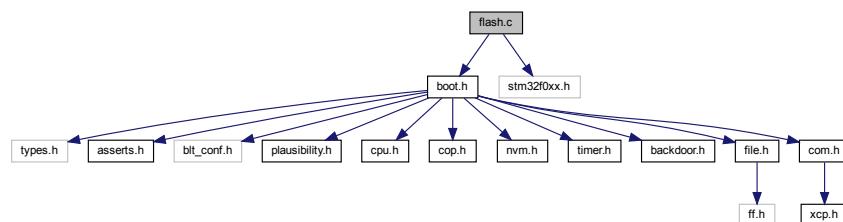
Referenced by [FlashEraseSectors\(\)](#), [FlashGetSector\(\)](#), [FlashGetSectorBaseAddr\(\)](#), [FlashGetSectorSize\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.53 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32f0xx.h"

Include dependency graph for ARMCM0_STM32F0/flash.c:
```



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR** (0xff)
Value for an invalid flash sector.
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SECTORS** (sizeof(flashLayout)/sizeof(flashLayout[0]))
Total numbers of sectors in array flashLayout[].
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **FLASH_ERASE_BLOCK_SIZE** (**FLASH_PAGE_SIZE**)
Number of bytes to erase per erase operation.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0xC0)
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- static **blt_bool FlashInitBlock** (**tFlashBlockInfo** *block, **blt_addr** address)
Copies data currently in flash to the block->data and sets the base address.
- static **tFlashBlockInfo** * **FlashSwitchBlock** (**tFlashBlockInfo** *block, **blt_addr** base_addr)
Switches blocks by programming the current one and initializing the next.
- static **blt_bool FlashAddToBlock** (**tFlashBlockInfo** *block, **blt_addr** address, **blt_int8u** *data, **blt_int32u** len)
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static **blt_bool FlashWriteBlock** (**tFlashBlockInfo** *block)
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static **blt_bool FlashEraseSectors** (**blt_int8u** first_sector, **blt_int8u** last_sector)
Erases the flash sectors from first_sector up until last_sector.
- static **blt_int8u FlashGetSector** (**blt_addr** address)
Determines the flash sector the address is in.
- static **blt_addr FlashGetSectorBaseAddr** (**blt_int8u** sector)
Determines the flash sector base address.
- static **blt_addr FlashGetSectorSize** (**blt_int8u** sector)
Determines the flash sector size.
- void **FlashInit** (void)
Initializes the flash driver.
- **blt_bool FlashWrite** (**blt_addr** addr, **blt_int32u** len, **blt_int8u** *data)

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

- `blt_bool FlashWriteChecksum (void)`

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

- `blt_bool FlashVerifyChecksum (void)`

Verifies the checksum, which indicates that a valid user program is present and can be started.

- `blt_bool FlashDone (void)`

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

- `blt_addr Flash GetUserProgBaseAddress (void)`

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`

If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to > 0 in `blt_conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.

- static `tFlashBlockInfo blockInfo`

Local variable with information about the flash block that is currently being operated on.

- static `tFlashBlockInfo bootBlockInfo`

Local variable with information about the flash boot block.

7.53.1 Detailed Description

Bootloader flash driver source file.

7.53.2 Function Documentation

7.53.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.53.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.53.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.53.2.4 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector,
    blt_int8u last_sector ) [static]
```

Erases the flash sectors from *first_sector* up until *last_sector*.

Parameters

| | |
|---------------------|----------------------------|
| <i>first_sector</i> | First flash sector number. |
| <i>last_sector</i> | Last flash sector number. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.53.2.5 FlashGetSector()

```
static blt_int8u FlashGetSector (
    blt_addr address ) [static]
```

Determines the flash sector the address is in.

Parameters

| | |
|----------------|------------------------------|
| <i>address</i> | Address in the flash sector. |
|----------------|------------------------------|

Returns

Flash sector number or FLASH_INVALID_SECTOR.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.53.2.6 FlashGetSectorBaseAddr()

```
static blt_addr FlashGetSectorBaseAddr (
    blt_int8u sector ) [static]
```

Determines the flash sector base address.

Parameters

| | |
|---------------|------------------------------------|
| <i>sector</i> | Sector to get the base address of. |
|---------------|------------------------------------|

Returns

Flash sector base address or FLASH_INVALID_ADDRESS.

Referenced by [FlashEraseSectors\(\)](#).

7.53.2.7 FlashGetSectorSize()

```
static blt_addr FlashGetSectorSize (
    blt_int8u sector ) [static]
```

Determines the flash sector size.

Parameters

| | |
|---------------|----------------------------|
| <i>sector</i> | Sector to get the size of. |
|---------------|----------------------------|

Returns

Flash sector size or 0.

Referenced by [FlashEraseSectors\(\)](#).

7.53.2.8 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.53.2.9 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.53.2.10 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.53.2.11 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.53.2.12 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.53.2.13 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.53.2.14 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.53.2.15 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.53.3 Variable Documentation

7.53.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.53.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.53.3.3 flashLayout

```
const tFlashSector flashLayout[ ] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory.

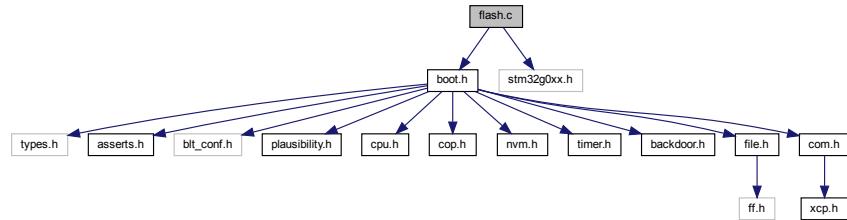
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten. The current flash layout does not reflect the minimum sector size of the physical flash (1 - 2kb), because this would make the table quit long and a waste of ROM. The minimum sector size is only really needed when erasing the flash. This can still be done in combination with macro FLASH_ERASE_BLOCK_SIZE. Note that the term sector here is used in a different meaning than in the controller's reference manual.

Referenced by [FlashEraseSectors\(\)](#), [FlashGetSector\(\)](#), [FlashGetSectorBaseAddr\(\)](#), [FlashGetSectorSize\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.54 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32g0xx.h"
Include dependency graph for ARMCM0_STM32G0/flash.c:
```



Data Structures

- struct [tFlashSector](#)
Flash sector descriptor type.
- struct [tFlashBlockInfo](#)
Structure type for grouping flash block information.

Macros

- `#define FLASH_INVALID_SECTOR (0xff)`
Value for an invalid sector number.
- `#define FLASH_INVALID_ADDRESS (0xffffffff)`
Value for an invalid flash address.
- `#define FLASH_WRITE_BLOCK_SIZE (512)`
Standard size of a flash block for writing.
- `#define FLASH_TOTAL_SECTORS (sizeof(flashLayout)/sizeof(flashLayout[0]))`
Total numbers of sectors in array flashLayout[].
- `#define FLASH_END_ADDRESS`
End address of the bootloader programmable flash.
- `#define FLASH_ERASE_BLOCK_SIZE (FLASH_PAGE_SIZE)`
Hardware erase unit size (sectors must be multiples of this)
- `#define BOOT_FLASH_VECTOR_TABLE_CS_OFFSET (0xBC)`
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overriden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- `static blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- `static tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- `static blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- `static blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- `static blt_bool FlashEraseSectors (blt_int8u first_sector, blt_int8u last_sector)`
Erases the flash sectors from first_sector up until last_sector.
- `static blt_int8u FlashGetSector (blt_addr address)`
Determines the flash sector the address is in.
- `static blt_addr FlashGetSectorBaseAddr (blt_int8u sector)`
Determines the flash sector base address.
- `static blt_addr FlashGetSectorSize (blt_int8u sector)`
Determines the flash sector size.
- `static blt_int32u FlashGetBank (blt_addr address)`
Determines the flash bank that the address belongs to.
- `static blt_int32u FlashGetPage (blt_addr address)`
Determines the flash page that the address belongs to.
- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

- **blt_bool FlashWriteChecksum (void)**
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- **blt_bool FlashVerifyChecksum (void)**
Verifies the checksum, which indicates that a valid user program is present and can be started.
- **blt_bool FlashDone (void)**
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- **blt_addr Flash GetUserProgBaseAddress (void)**
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const **tFlashSector flashLayout []**
If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to > 0 in `blt_conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.
- static **tFlashBlockInfo blockInfo**
Local variable with information about the flash block that is currently being operated on.
- static **tFlashBlockInfo bootBlockInfo**
Local variable with information about the flash boot block.

7.54.1 Detailed Description

Bootloader flash driver source file.

7.54.2 Function Documentation

7.54.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.54.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.54.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.54.2.4 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector,
    blt_int8u last_sector ) [static]
```

Erases the flash sectors from *first_sector* up until *last_sector*.

Parameters

| | |
|---------------------|----------------------------|
| <i>first_sector</i> | First flash sector number. |
| <i>last_sector</i> | Last flash sector number. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.54.2.5 FlashGetBank()

```
static blt_int32u FlashGetBank (
    blt_addr address ) [static]
```

Determines the flash bank that the address belongs to.

Parameters

| | |
|----------------|-----------------------|
| <i>address</i> | Flash memory address. |
|----------------|-----------------------|

Returns

FLASH_BANK_1 if the address belongs to bank 1, FLASH_BANK_2 otherwise.

Referenced by [FlashEraseSectors\(\)](#), and [FlashGetPage\(\)](#).

7.54.2.6 FlashGetPage()

```
static blt_int32u FlashGetPage (
    blt_addr address ) [static]
```

Determines the flash page that the address belongs to.

end of FlashGetBank

Parameters

| | |
|----------------|-----------------------|
| <i>address</i> | Flash memory address. |
|----------------|-----------------------|

Returns

Page number.

Referenced by [FlashEraseSectors\(\)](#).

7.54.2.7 FlashGetSector()

```
static blt_int8u FlashGetSector (
    blt_addr address ) [static]
```

Determines the flash sector the address is in.

Parameters

| | |
|----------------------|------------------------------|
| <code>address</code> | Address in the flash sector. |
|----------------------|------------------------------|

Returns

Flash sector number or FLASH_INVALID_SECTOR.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.54.2.8 FlashGetSectorBaseAddr()

```
static blt_addr FlashGetSectorBaseAddr (
    blt_int8u sector ) [static]
```

Determines the flash sector base address.

Parameters

| | |
|---------------------|------------------------------------|
| <code>sector</code> | Sector to get the base address of. |
|---------------------|------------------------------------|

Returns

Flash sector base address or FLASH_INVALID_ADDRESS.

Referenced by [FlashEraseSectors\(\)](#).

7.54.2.9 FlashGetSectorSize()

```
static blt_addr FlashGetSectorSize (
    blt_int8u sector ) [static]
```

Determines the flash sector size.

Parameters

| | |
|---------------------|----------------------------|
| <code>sector</code> | Sector to get the size of. |
|---------------------|----------------------------|

Returns

Flash sector size or 0.

Referenced by [FlashEraseSectors\(\)](#).

7.54.2.10 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.54.2.11 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.54.2.12 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.54.2.13 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is now being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.54.2.14 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.54.2.15 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.54.2.16 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.54.2.17 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.54.3 Variable Documentation

7.54.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.54.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way it is possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.54.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory.

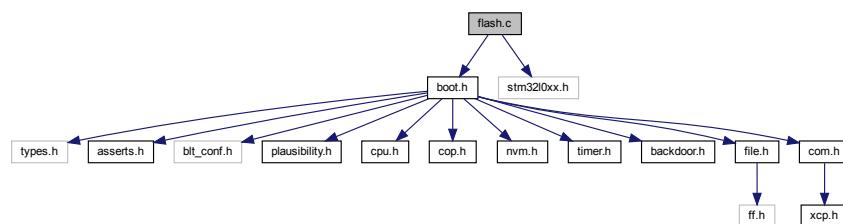
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten. The current flash layout does not reflect the minimum sector size of the physical flash (2 KiB), because this would make the table quite long and a waste of ROM. The minimum sector size is only really needed when erasing the flash. This can still be done in combination with macro FLASH_ERASE_BLOCK_SIZE. Note that the term sector here is used in a different meaning than in the controller's reference manual.

Referenced by [FlashEraseSectors\(\)](#), [FlashGetSector\(\)](#), [FlashGetSectorBaseAddr\(\)](#), [FlashGetSectorSize\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.55 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32l0xx.h"
Include dependency graph for ARMCM0_STM32L0/flash.c:
```



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SEGMENTS** (sizeof(**flashLayout**)/sizeof(**flashLayout**[0]))
*Total numbers of segments in array **flashLayout**[].*
- #define **FLASH_LAST_SEGMENT_IDX** (**FLASH_TOTAL_SEGMENTS**-1)
*Index of the last segment in array **flashLayout**[].*
- #define **FLASH_START_ADDRESS** (**flashLayout**[0].sector_start)
Start address of the bootloader programmable flash.
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **FLASH_ERASE_BLOCK_SIZE** (2048)
Number of bytes to erase per erase operation. Usually this is set to the value as defined by hardware (FLASH_→ PAGE_SIZE). However, on the STM32L0, this value is less than FLASH_WRITE_BLOCK_SIZE, which would lead to problems. Therefore this macro was set to a value that is a multiple of both FLASH_WRITE_BLOCK_SIZE and FLASH_PAGE_SIZE.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0xC0)
*Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in **blt_conf.h**, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.*

Functions

- static `blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- static `tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- static `blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static `blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- void `FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- static `tFlashBlockInfo blockInfo`
Local variable with information about the flash block that is currently being operated on.
- static `tFlashBlockInfo bootBlockInfo`
Local variable with information about the flash boot block.

7.55.1 Detailed Description

Bootloader flash driver source file.

7.55.2 Function Documentation

7.55.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.55.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.55.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.55.2.4 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.55.2.5 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.55.2.6 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.55.2.7 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.55.2.8 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.55.2.9 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.55.2.10 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.55.2.11 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.55.3 Variable Documentation

7.55.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.55.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block needs to be written twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way it is possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.55.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory.

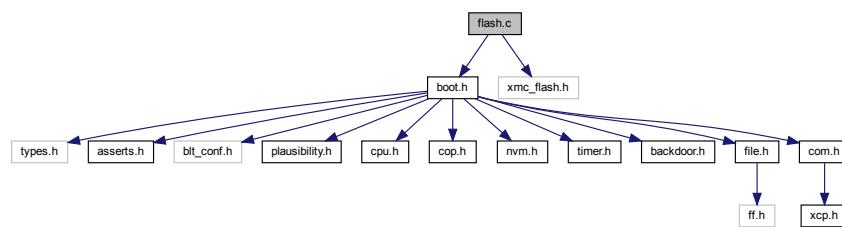
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten. The current flash layout does not reflect the minimum sector size of the physical flash (128 bytes), because this would make the table quite long and a waste of ROM. The minimum sector size is only really needed when erasing the flash. This can still be done in combination with macro FLASH_ERASE_BLOCK_SIZE.

Referenced by [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.56 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "xmc_flash.h"
Include dependency graph for ARMCM0_XMC1/flash.c:
```



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR** (0xff)
Value for an invalid flash sector.
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (256)
Standard size of a flash block for writing. It should be large enough so that the OpenBLT checksum fits in the first (boot) block. Note that this value is set to the hardware defined size of a XMC1 flash page.
- #define **FLASH_TOTAL_SECTORS** (sizeof(**flashLayout**)/sizeof(**flashLayout**[0]))
*Total numbers of sectors in array **flashLayout**[].*
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x01C)
*Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in **blt_conf.h**, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.*
- #define **FLASH_ERASE_TIME_MAX_MS** (10)
Maximum time for a sector erase operation as specified by the XCM1xxx data-sheet with an added 20% margin.
- #define **FLASH_PROGRAM_TIME_MAX_MS** (5)
Maximum time for a page program operation as specified by the XCM1xxx data-sheet with an added 20% margin.

Functions

- static `blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- static `tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- static `blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static `blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static `blt_bool FlashEraseSectors (blt_int8u first_sector, blt_int8u last_sector)`
Erases the flash sectors from first_sector up until last_sector.
- static `blt_int8u FlashGetSector (blt_addr address)`
Determines the flash sector the address is in.
- static `blt_addr FlashGetSectorBaseAddr (blt_int8u sector)`
Obtains the base address of the specified sector.
- void `FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- static `tFlashBlockInfo blockInfo`
Local variable with information about the flash block that is currently being operated on.
- static `tFlashBlockInfo bootBlockInfo`
Local variable with information about the flash boot block.

7.56.1 Detailed Description

Bootloader flash driver source file.

7.56.2 Function Documentation

7.56.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.56.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.56.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.56.2.4 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector,
    blt_int8u last_sector ) [static]
```

Erases the flash sectors from *first_sector* up until *last_sector*.

Parameters

| | |
|---------------------|----------------------------|
| <i>first_sector</i> | First flash sector number. |
| <i>last_sector</i> | Last flash sector number. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.56.2.5 FlashGetSector()

```
static blt_int8u FlashGetSector (
    blt_addr address ) [static]
```

Determines the flash sector the *address* is in.

Parameters

| | |
|----------------|------------------------------|
| <i>address</i> | Address in the flash sector. |
|----------------|------------------------------|

Returns

Flash sector number or FLASH_INVALID_SECTOR.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.56.2.6 FlashGetSectorBaseAddr()

```
static blt_addr FlashGetSectorBaseAddr (
    blt_int8u sector ) [static]
```

Obtains the base address of the specified sector.

Parameters

| | |
|--------|------------------------------------|
| sector | Sector to get the base address of. |
|--------|------------------------------------|

Returns

Base Base address of the sector if found, FLASH_INVALID_ADDRESS otherwise.

Referenced by [FlashEraseSectors\(\)](#).

7.56.2.7 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.56.2.8 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.56.2.9 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.56.2.10 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.56.2.11 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.56.2.12 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.56.2.13 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.56.2.14 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.56.3 Variable Documentation

7.56.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.56.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way it is possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.56.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory.

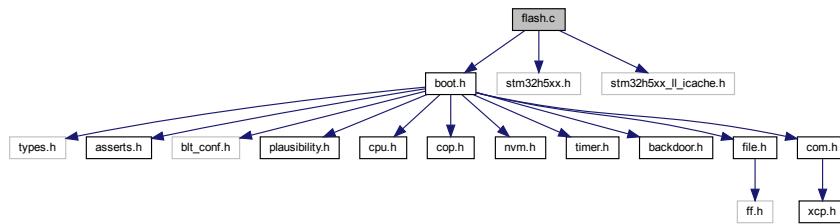
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten. Note that the table contains uncached addresses, because flash program/ erase operations need to be performed on uncached addresses. This flash driver automatically translates cached to uncached addresses, so there is no need for the user to adjust this when calling this driver's API.

Referenced by [FlashEraseSectors\(\)](#), [FlashGetSector\(\)](#), [FlashGetSectorBaseAddr\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.57 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32h5xx.h"
#include "stm32h5xx_ll_icache.h"
Include dependency graph for ARCM33_STM32H5/flash.c:
```



Data Structures

- struct [tFlashSector](#)
Flash sector descriptor type.
- struct [tFlashBlockInfo](#)
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR_IDX** (0xff)
Value for an invalid sector entry index into flashLayout[].
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (1024)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SECTORS** (sizeof([flashLayout](#))/sizeof([flashLayout\[0\]](#)))
Total numbers of sectors in array flashLayout[].
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x24C)
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- static `blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`

Copies data currently in flash to the block->data and sets the base address.
- static `tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`

Switches blocks by programming the current one and initializing the next.
- static `blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static `blt_bool FlashWriteBlock (tFlashBlockInfo *block)`

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static `blt_bool FlashEmptyCheckSector (blt_int8u sector_idx)`

Checks if the flash sector is already completely erased.
- static `blt_bool FlashEraseSectors (blt_int8u first_sector_idx, blt_int8u last_sector_idx)`

Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.
- static `blt_int8u FlashGetSectorIdx (blt_addr address)`

Determines the index into the flashLayout[] array of the flash sector that the specified address is in.
- static `blt_int32u FlashGetBank (blt_addr address)`

Determines the flash bank that the address belongs to.
- static `blt_int32u FlashGetPage (blt_addr address)`

Determines the flash page that the address belongs to.
- void `FlashInit (void)`

Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`

Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- static `tFlashBlockInfo blockInfo`

Local variable with information about the flash block that is currently being operated on.
- static `tFlashBlockInfo bootBlockInfo`

Local variable with information about the flash boot block.

7.57.1 Detailed Description

Bootloader flash driver source file.

7.57.2 Function Documentation

7.57.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.57.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.57.2.3 FlashEmptyCheckSector()

```
static blt_bool FlashEmptyCheckSector (
    blt_int8u sector_idx ) [static]
```

Checks if the flash sector is already completely erased.

Parameters

| | |
|-------------------------|---|
| <code>sector_idx</code> | flash sector number index into flashLayout[]. |
|-------------------------|---|

Returns

BLT_TRUE if the flash sector is already erased, BLT_FALSE otherwise.

Referenced by [FlashEraseSectors\(\)](#).

7.57.2.4 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------------|------------------|
| <code>addr</code> | Start address. |
| <code>len</code> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.57.2.5 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector_idx,
    blt_int8u last_sector_idx ) [static]
```

Erases the flash sectors from indices `first_sector_idx` up until `last_sector_idx` into the `flashLayout[]` array.

Parameters

| | |
|-------------------------------|---|
| <code>first_sector_idx</code> | First flash sector number index into flashLayout[]. |
| <code>last_sector_idx</code> | Last flash sector number index into flashLayout[]. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.57.2.6 FlashGetBank()

```
static blt_int32u FlashGetBank (
    blt_addr address ) [static]
```

Determines the flash bank that the address belongs to.

Parameters

| | |
|----------------------|-----------------------|
| <code>address</code> | Flash memory address. |
|----------------------|-----------------------|

Returns

`FLASH_BANK_1` if the address belongs to bank 1, `FLASH_BANK_2` otherwise.

Referenced by [FlashEraseSectors\(\)](#), and [FlashGetPage\(\)](#).

7.57.2.7 FlashGetPage()

```
static blt_int32u FlashGetPage (
    blt_addr address ) [static]
```

Determines the flash page that the address belongs to.

end of FlashGetBank

Parameters

| | |
|----------------------|-----------------------|
| <code>address</code> | Flash memory address. |
|----------------------|-----------------------|

Returns

Page number.

Referenced by [FlashEraseSectors\(\)](#).

7.57.2.8 FlashGetSectorIdx()

```
static blt_int8u FlashGetSectorIdx (
    blt_addr address ) [static]
```

Determines the index into the `flashLayout[]` array of the flash sector that the specified address is in.

Parameters

| | |
|----------------------|------------------------------|
| <code>address</code> | Address in the flash sector. |
|----------------------|------------------------------|

Returns

Flash sector index in `flashLayout[]` or `FLASH_INVALID_SECTOR_IDX`.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.57.2.9 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the `flashLayout` table.

Returns

Base address.

7.57.2.10 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Attention

This entire flash driver should not use the `FLASH_SIZE` or `FLASH_BANK_SIZE` macros. These macros indirectly access `FLASHSIZE_BASE` (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro `BOOT_NVM_SIZE_KB` instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the `FLASH_SIZE` and `FLASH_BANK_SIZE` macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.57.2.11 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the `block->data` and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.57.2.12 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is now being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.57.2.13 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.57.2.14 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.57.2.15 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.57.2.16 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.57.3 Variable Documentation

7.57.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.57.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way it is possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.57.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory.

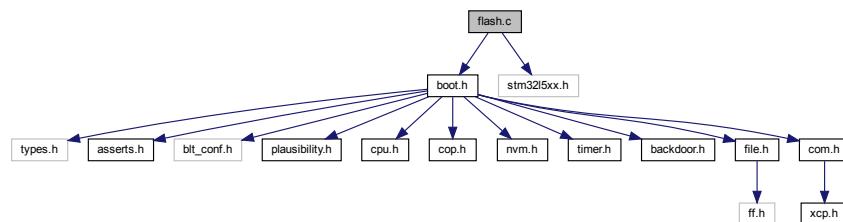
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

Referenced by [FlashEmptyCheckSector\(\)](#), [FlashEraseSectors\(\)](#), [FlashGetSectorIdx\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.58 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32l5xx.h"
Include dependency graph for ARMCM33_STM32L5/flash.c:
```



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR_IDX** (0xff)
Value for an invalid sector entry index into flashLayout[].
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SECTORS** (sizeof(flashLayout)/sizeof(flashLayout[0]))
Total numbers of sectors in array flashLayout[].
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x1F4)
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- static `blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- static `tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- static `blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static `blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static `blt_bool FlashEraseSectors (blt_int8u first_sector_idx, blt_int8u last_sector_idx)`
Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.
- static `blt_int8u FlashGetSectorIdx (blt_addr address)`
Determines the index into the flashLayout[] array of the flash sector that the specified address is in.
- static `blt_int32u FlashGetPageSize (void)`
Determines the size of a flash pages, defined by hardware. This also defines the minimal erase size.
- static `blt_bool FlashIsDualBankMode (void)`
Determines if the flash device is configured as dual bank mode or single bank mode.
- static `blt_int32u FlashGetBank (blt_addr address)`
Determines the flash bank that the address belongs to.
- static `blt_int32u FlashGetPage (blt_addr address)`
Determines the flash page that the address belongs to.
- void `FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- static `tFlashBlockInfo blockInfo`
Local variable with information about the flash block that is currently being operated on.
- static `tFlashBlockInfo bootBlockInfo`
Local variable with information about the flash boot block.

7.58.1 Detailed Description

Bootloader flash driver source file.

7.58.2 Function Documentation

7.58.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.58.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.58.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.58.2.4 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector_idx,
    blt_int8u last_sector_idx ) [static]
```

Erases the flash sectors from indices *first_sector_idx* up until *last_sector_idx* into the *flashLayout[]* array.

Parameters

| | |
|-------------------------|---|
| <i>first_sector_idx</i> | First flash sector number index into <i>flashLayout[]</i> . |
| <i>last_sector_idx</i> | Last flash sector number index into <i>flashLayout[]</i> . |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.58.2.5 FlashGetBank()

```
static blt_int32u FlashGetBank (
    blt_addr address ) [static]
```

Determines the flash bank that the address belongs to.

Parameters

| | |
|----------------------|-----------------------|
| <code>address</code> | Flash memory address. |
|----------------------|-----------------------|

Returns

`FLASH_BANK_1` if the address belongs to bank 1, `FLASH_BANK_2` otherwise.

Referenced by [FlashEraseSectors\(\)](#), and [FlashGetPage\(\)](#).

7.58.2.6 FlashGetPage()

```
static blt_int32u FlashGetPage (
    blt_addr address ) [static]
```

Determines the flash page that the address belongs to.

end of FlashGetBank

Parameters

| | |
|----------------------|-----------------------|
| <code>address</code> | Flash memory address. |
|----------------------|-----------------------|

Returns

Page number.

Referenced by [FlashEraseSectors\(\)](#).

7.58.2.7 FlashGetPageSize()

```
static blt_int32u FlashGetPageSize (
    void ) [static]
```

Determines the size of a flash pages, defined by hardware. This also defines the minimal erase size.

Returns

Size of a flash page.

Referenced by [FlashEraseSectors\(\)](#), and [FlashGetPage\(\)](#).

7.58.2.8 FlashGetSectorIdx()

```
static blt_int8u FlashGetSectorIdx (
    blt_addr address ) [static]
```

Determines the index into the `flashLayout[]` array of the flash sector that the specified address is in.

Parameters

| | |
|----------------------|------------------------------|
| <code>address</code> | Address in the flash sector. |
|----------------------|------------------------------|

Returns

Flash sector index in `flashLayout[]` or `FLASH_INVALID_SECTOR_IDX`.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.58.2.9 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the `flashLayout` table.

Returns

Base address.

7.58.2.10 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.58.2.11 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the `block->data` and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.58.2.12 FlashIsDualBankMode()

```
static blt_bool FlashIsDualBankMode (
    void ) [static]
```

Determines if the flash device is configured as dual bank mode or single bank mode.

Returns

BLT_TRUE if configured as dual bank mode, BLT_FALSE for single bank mode.

Referenced by [FlashGetBank\(\)](#), [FlashGetPage\(\)](#), and [FlashGetPageSize\(\)](#).

7.58.2.13 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is now being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.58.2.14 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.58.2.15 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.58.2.16 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.58.2.17 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.58.3 Variable Documentation

7.58.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.58.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way it is possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.58.3.3 flashLayout

```
const tFlashSector flashLayout[ ] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array wit the layout of the flash memory.

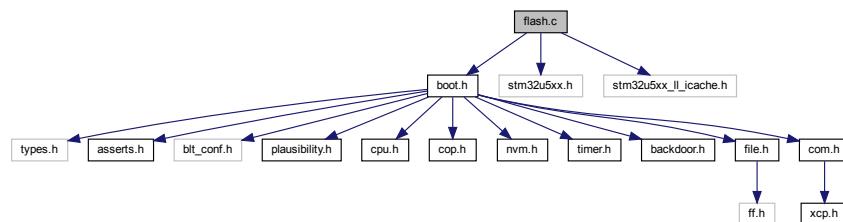
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

Referenced by [FlashEraseSectors\(\)](#), [FlashGetSectorIdx\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.59 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32u5xx.h"
#include "stm32u5xx_ll_icache.h"
Include dependency graph for ARMCM33_STM32U5/flash.c:
```



Data Structures

- struct [tFlashSector](#)
Flash sector descriptor type.
- struct [tFlashBlockInfo](#)
Structure type for grouping flash block information.

Macros

- `#define FLASH_INVALID_SECTOR_IDX (0xff)`
Value for an invalid sector entry index into flashLayout[].
- `#define FLASH_INVALID_ADDRESS (0xffffffff)`
Value for an invalid flash address.
- `#define FLASH_WRITE_BLOCK_SIZE (1024)`
Standard size of a flash block for writing.
- `#define FLASH_TOTAL_SECTORS (sizeof(flashLayout)/sizeof(flashLayout[0]))`
Total numbers of sectors in array flashLayout[].
- `#define FLASH_END_ADDRESS`
End address of the bootloader programmable flash.
- `#define BOOT_FLASH_VECTOR_TABLE_CS_OFFSET (0x234)`
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- `static blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- `static tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- `static blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- `static blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- `static blt_bool FlashEmptyCheckSector (blt_int8u sector_idx)`
Checks if the flash sector is already completely erased.
- `static blt_bool FlashEraseSectors (blt_int8u first_sector_idx, blt_int8u last_sector_idx)`
Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.
- `static blt_int8u FlashGetSectorIdx (blt_addr address)`
Determines the index into the flashLayout[] array of the flash sector that the specified address is in.
- `static blt_int32u FlashGetBank (blt_addr address)`
Determines the flash bank that the address belongs to.
- `static blt_int32u FlashGetPage (blt_addr address)`
Determines the flash page that the address belongs to.
- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`

Verifies the checksum, which indicates that a valid user program is present and can be started.

- **blt_bool FlashDone (void)**

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

- **blt_addr Flash GetUserProgBaseAddress (void)**

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const **tFlashSector flashLayout []**

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

- static **tFlashBlockInfo blockInfo**

Local variable with information about the flash block that is currently being operated on.

- static **tFlashBlockInfo bootBlockInfo**

Local variable with information about the flash boot block.

7.59.1 Detailed Description

Bootloader flash driver source file.

7.59.2 Function Documentation

7.59.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.59.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.59.2.3 FlashEmptyCheckSector()

```
static blt_bool FlashEmptyCheckSector (
    blt_int8u sector_idx ) [static]
```

Checks if the flash sector is already completely erased.

Parameters

| | |
|-------------------------|---|
| <code>sector_idx</code> | flash sector number index into <code>flashLayout[]</code> . |
|-------------------------|---|

Returns

BLT_TRUE if the flash sector is already erased, BLT_FALSE otherwise.

Referenced by [FlashEraseSectors\(\)](#).

7.59.2.4 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.59.2.5 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector_idx,
    blt_int8u last_sector_idx ) [static]
```

Erases the flash sectors from indices *first_sector_idx* up until *last_sector_idx* into the *flashLayout[]* array.

Parameters

| | |
|-------------------------|---|
| <i>first_sector_idx</i> | First flash sector number index into <i>flashLayout[]</i> . |
| <i>last_sector_idx</i> | Last flash sector number index into <i>flashLayout[]</i> . |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.59.2.6 FlashGetBank()

```
static blt_int32u FlashGetBank (
    blt_addr address ) [static]
```

Determines the flash bank that the address belongs to.

Parameters

| | |
|----------------|-----------------------|
| <i>address</i> | Flash memory address. |
|----------------|-----------------------|

Returns

FLASH_BANK_1 if the address belongs to bank 1, FLASH_BANK_2 otherwise.

Referenced by [FlashEraseSectors\(\)](#), and [FlashGetPage\(\)](#).

7.59.2.7 FlashGetPage()

```
static blt_int32u FlashGetPage (
    blt_addr address ) [static]
```

Determines the flash page that the address belongs to.

end of FlashGetBank

Parameters

| | |
|----------------------|-----------------------|
| <code>address</code> | Flash memory address. |
|----------------------|-----------------------|

Returns

Page number.

Referenced by [FlashEraseSectors\(\)](#).

7.59.2.8 FlashGetSectorIdx()

```
static blt_int8u FlashGetSectorIdx (
    blt_addr address ) [static]
```

Determines the index into the `flashLayout[]` array of the flash sector that the specified address is in.

Parameters

| | |
|----------------------|------------------------------|
| <code>address</code> | Address in the flash sector. |
|----------------------|------------------------------|

Returns

Flash sector index in `flashLayout[]` or `FLASH_INVALID_SECTOR_IDX`.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.59.2.9 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the `flashLayout` table.

Returns

Base address.

7.59.2.10 FlashInit()

```
void FlashInit (
    void  )
```

Initializes the flash driver.

Returns

none.

7.59.2.11 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.59.2.12 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is now being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.59.2.13 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.59.2.14 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.59.2.15 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.59.2.16 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.59.3 Variable Documentation

7.59.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.59.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also the where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.59.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array wit the layout of the flash memory.

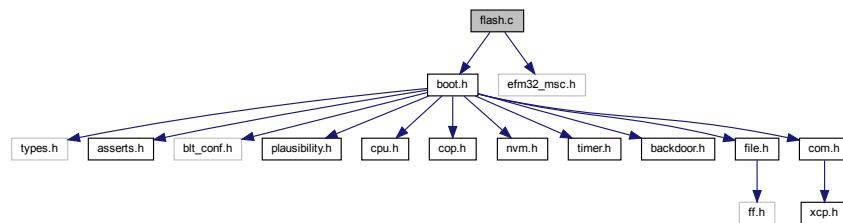
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

Referenced by [FlashEmptyCheckSector\(\)](#), [FlashEraseSectors\(\)](#), [FlashGetSectorIdx\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.60 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "efm32_msc.h"
Include dependency graph for ARMCM3_EFM32/flash.c:
```



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR** (0xff)
Value for an invalid flash sector.
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SECTORS** (sizeof(flashLayout)/sizeof(flashLayout[0]))
Total numbers of sectors in array flashLayout[].
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x0B8)
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- static **blt_bool FlashInitBlock** (**tFlashBlockInfo** *block, **blt_addr** address)
Copies data currently in flash to the block->data and sets the base address.
- static **tFlashBlockInfo** * **FlashSwitchBlock** (**tFlashBlockInfo** *block, **blt_addr** base_addr)
Switches blocks by programming the current one and initializing the next.
- static **blt_bool FlashAddToBlock** (**tFlashBlockInfo** *block, **blt_addr** address, **blt_int8u** *data, **blt_int32u** len)
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static **blt_bool FlashWriteBlock** (**tFlashBlockInfo** *block)
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static **blt_bool FlashEraseSectors** (**blt_int8u** first_sector, **blt_int8u** last_sector)
Erases the flash sectors from first_sector up until last_sector.
- static **blt_int8u FlashGetSector** (**blt_addr** address)
Determines the flash sector the address is in.
- static **blt_addr FlashGetSectorBaseAddr** (**blt_int8u** sector)
Determines the flash sector base address.
- static **blt_addr FlashGetSectorSize** (**blt_int8u** sector)
Determines the flash sector size.
- static **blt_int32u FlashCalcPageSize** (**void**)
Determines the flash page size for the specific EFM32 derivative. This is the minimum erase size.
- void **FlashInit** (**void**)
Initializes the flash driver.
- **blt_bool FlashWrite** (**blt_addr** addr, **blt_int32u** len, **blt_int8u** *data)

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

- `blt_bool FlashWriteChecksum (void)`

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

- `blt_bool FlashVerifyChecksum (void)`

Verifies the checksum, which indicates that a valid user program is present and can be started.

- `blt_bool FlashDone (void)`

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

- `blt_addr Flash GetUserProgBaseAddress (void)`

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`

If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to > 0 in `blt_conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.

- static `tFlashBlockInfo blockInfo`

Local variable with information about the flash block that is currently being operated on.

- static `tFlashBlockInfo bootBlockInfo`

Local variable with information about the flash boot block.

7.60.1 Detailed Description

Bootloader flash driver source file.

7.60.2 Function Documentation

7.60.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.60.2.2 FlashCalcPageSize()

```
static blt_int32u FlashCalcPageSize (
    void ) [static]
```

Determines the flash page size for the specific EFM32 derivative. This is the minimum erase size.

Returns

The flash page size.

Referenced by [FlashEraseSectors\(\)](#).

7.60.2.3 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.60.2.4 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.60.2.5 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector,
    blt_int8u last_sector ) [static]
```

Erases the flash sectors from *first_sector* up until *last_sector*.

Parameters

| | |
|---------------------|----------------------------|
| <i>first_sector</i> | First flash sector number. |
| <i>last_sector</i> | Last flash sector number. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.60.2.6 FlashGetSector()

```
static blt_int8u FlashGetSector (
    blt_addr address ) [static]
```

Determines the flash sector the *address* is in.

Parameters

| | |
|----------------|------------------------------|
| <i>address</i> | Address in the flash sector. |
|----------------|------------------------------|

Returns

Flash sector number or FLASH_INVALID_SECTOR.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.60.2.7 FlashGetSectorBaseAddr()

```
static blt_addr FlashGetSectorBaseAddr (
    blt_int8u sector ) [static]
```

Determines the flash sector base address.

Parameters

| | |
|---------------|------------------------------------|
| <i>sector</i> | Sector to get the base address of. |
|---------------|------------------------------------|

Returns

Flash sector base address or FLASH_INVALID_ADDRESS.

Referenced by [FlashEraseSectors\(\)](#).

7.60.2.8 FlashGetSectorSize()

```
static blt_addr FlashGetSectorSize (
    blt_int8u sector ) [static]
```

Determines the flash sector size.

Parameters

| | |
|---------------|----------------------------|
| <i>sector</i> | Sector to get the size of. |
|---------------|----------------------------|

Returns

Flash sector size or 0.

Referenced by [FlashEraseSectors\(\)](#).

7.60.2.9 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.60.2.10 FlashInit()

```
void FlashInit (
    void  )
```

Initializes the flash driver.

Returns

none.

7.60.2.11 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.60.2.12 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.60.2.13 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.60.2.14 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.60.2.15 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.60.2.16 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.60.3 Variable Documentation

7.60.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.60.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also the where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.60.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array wit the layout of the flash memory.

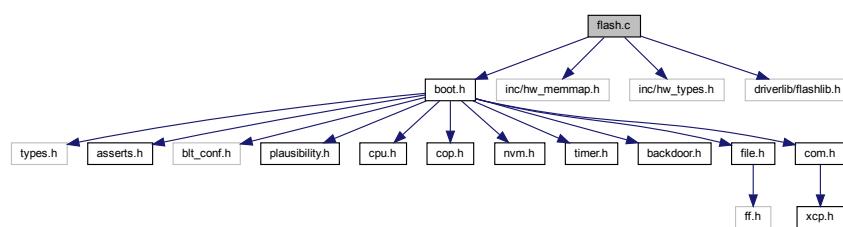
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten. The current flash layout does not reflect the minimum sector size of the physical flash (1 - 2kb), because this would make the table quit long and a waste of ROM. The minimum sector size is only really needed when erasing the flash. This can still be done in combination with macro FLASH_ERASE_BLOCK_SIZE.

Referenced by [FlashEraseSectors\(\)](#), [FlashGetSector\(\)](#), [FlashGetSectorBaseAddr\(\)](#), [FlashGetSectorSize\(\)](#), [FlashGetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.61 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/flashlib.h"
Include dependency graph for ARCMC3_LM3S/flash.c:
```



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR** (0xff)
Value for an invalid flash sector.
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SECTORS** (sizeof(flashLayout)/sizeof(flashLayout[0]))
Total numbers of sectors in array flashLayout[].
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **FLASH_ERASE_BLOCK_SIZE** (0x400)
Number of bytes to erase per erase operation.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0xF0)
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- static **blt_bool FlashInitBlock** (**tFlashBlockInfo** *block, **blt_addr** address)
Copies data currently in flash to the block->data and sets the base address.
- static **tFlashBlockInfo** * **FlashSwitchBlock** (**tFlashBlockInfo** *block, **blt_addr** base_addr)
Switches blocks by programming the current one and initializing the next.
- static **blt_bool FlashAddToBlock** (**tFlashBlockInfo** *block, **blt_addr** address, **blt_int8u** *data, **blt_int32u** len)
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static **blt_bool FlashWriteBlock** (**tFlashBlockInfo** *block)
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static **blt_bool FlashEraseSectors** (**blt_int8u** first_sector, **blt_int8u** last_sector)
Erases the flash sectors from first_sector up until last_sector.
- static **blt_int8u FlashGetSector** (**blt_addr** address)
Determines the flash sector the address is in.
- static **blt_addr FlashGetSectorBaseAddr** (**blt_int8u** sector)
Determines the flash sector base address.
- static **blt_addr FlashGetSectorSize** (**blt_int8u** sector)
Determines the flash sector size.
- void **FlashInit** (**void**)
Initializes the flash driver.
- **blt_bool FlashWrite** (**blt_addr** addr, **blt_int32u** len, **blt_int8u** *data)

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

- `blt_bool FlashWriteChecksum (void)`

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

- `blt_bool FlashVerifyChecksum (void)`

Verifies the checksum, which indicates that a valid user program is present and can be started.

- `blt_bool FlashDone (void)`

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

- `blt_addr Flash GetUserProgBaseAddress (void)`

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`

If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to > 0 in `blt_conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.

- static `tFlashBlockInfo blockInfo`

Local variable with information about the flash block that is currently being operated on.

- static `tFlashBlockInfo bootBlockInfo`

Local variable with information about the flash boot block.

7.61.1 Detailed Description

Bootloader flash driver source file.

7.61.2 Function Documentation

7.61.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.61.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.61.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.61.2.4 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector,
    blt_int8u last_sector ) [static]
```

Erases the flash sectors from *first_sector* up until *last_sector*.

Parameters

| | |
|---------------------|----------------------------|
| <i>first_sector</i> | First flash sector number. |
| <i>last_sector</i> | Last flash sector number. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.61.2.5 FlashGetSector()

```
static blt_int8u FlashGetSector (
    blt_addr address ) [static]
```

Determines the flash sector the address is in.

Parameters

| | |
|----------------|------------------------------|
| <i>address</i> | Address in the flash sector. |
|----------------|------------------------------|

Returns

Flash sector number or FLASH_INVALID_SECTOR.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.61.2.6 FlashGetSectorBaseAddr()

```
static blt_addr FlashGetSectorBaseAddr (
    blt_int8u sector ) [static]
```

Determines the flash sector base address.

Parameters

| | |
|---------------|------------------------------------|
| <i>sector</i> | Sector to get the base address of. |
|---------------|------------------------------------|

Returns

Flash sector base address or FLASH_INVALID_ADDRESS.

Referenced by [FlashEraseSectors\(\)](#).

7.61.2.7 FlashGetSectorSize()

```
static blt_addr FlashGetSectorSize (
    blt_int8u sector ) [static]
```

Determines the flash sector size.

Parameters

| | |
|---------------|----------------------------|
| <i>sector</i> | Sector to get the size of. |
|---------------|----------------------------|

Returns

Flash sector size or 0.

Referenced by [FlashEraseSectors\(\)](#).

7.61.2.8 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.61.2.9 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.61.2.10 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.61.2.11 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.61.2.12 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.61.2.13 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.61.2.14 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.61.2.15 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.61.3 Variable Documentation

7.61.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.61.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.61.3.3 flashLayout

```
const tFlashSector flashLayout[ ] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array wit the layout of the flash memory.

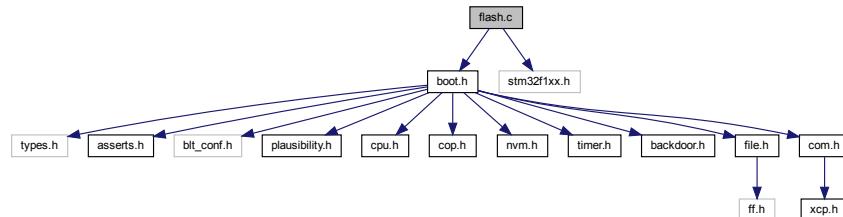
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten. The current flash layout does not reflect the minimum sector size of the physical flash (1 - 2kb), because this would make the table quit long and a waste of ROM. The minimum sector size is only really needed when erasing the flash. This can still be done in combination with macro FLASH_ERASE_BLOCK_SIZE.

Referenced by [FlashEraseSectors\(\)](#), [FlashGetSector\(\)](#), [FlashGetSectorBaseAddr\(\)](#), [FlashGetSectorSize\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.62 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32f1xx.h"
Include dependency graph for ARMCM3_STM32F1/flash.c:
```



Data Structures

- struct [tFlashSector](#)
Flash sector descriptor type.
- struct [tFlashBlockInfo](#)
Structure type for grouping flash block information.

Macros

- `#define FLASH_INVALID_ADDRESS (0xffffffff)`
Value for an invalid flash address.
- `#define FLASH_WRITE_BLOCK_SIZE (512)`
Standard size of a flash block for writing.
- `#define FLASH_TOTAL_SEGMENTS (sizeof(flashLayout)/sizeof(flashLayout[0]))`
Total numbers of segments in array flashLayout[].
- `#define FLASH_LAST_SEGMENT_IDX (FLASH_TOTAL_SEGMENTS-1)`
Index of the last segment in array flashLayout[].
- `#define FLASH_START_ADDRESS (flashLayout[0].sector_start)`
Start address of the bootloader programmable flash.
- `#define FLASH_END_ADDRESS`
End address of the bootloader programmable flash.
- `#define FLASH_ERASE_BLOCK_SIZE (FLASH_PAGE_SIZE)`
Number of bytes to erase per erase operation.
- `#define BOOT_FLASH_VECTOR_TABLE_CS_OFFSET (0x150)`
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- `static blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- `static tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- `static blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- `static blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const tFlashSector flashLayout []

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt.conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

- static tFlashBlockInfo blockInfo

Local variable with information about the flash block that is currently being operated on.

- static tFlashBlockInfo bootBlockInfo

Local variable with information about the flash boot block.

7.62.1 Detailed Description

Bootloader flash driver source file.

7.62.2 Function Documentation

7.62.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.62.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.62.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.62.2.4 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.62.2.5 FlashInit()

```
void FlashInit (
    void  )
```

Initializes the flash driver.

Returns

none.

7.62.2.6 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.62.2.7 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.62.2.8 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.62.2.9 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.62.2.10 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------------|--|
| <code>block</code> | Pointer to flash block info structure to operate on. |
|--------------------|--|

Returns

`BLT_TRUE` if successful, `BLT_FALSE` otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.62.2.11 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

`BLT_TRUE` if successful, `BLT_FALSE` otherwise.

7.62.3 Variable Documentation

7.62.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is `FLASH_WRITE_BLOCK_SIZE`. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.62.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also the where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.62.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array wit the layout of the flash memory.

Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten. The current flash layout does not reflect the minimum sector size of the physical flash (1 - 2kb), because this would make the table quit long and a waste of ROM. The minimum sector size is only really needed when erasing the flash. This can still be done in combination with macro FLASH_ERASE_BLOCK_SIZE.

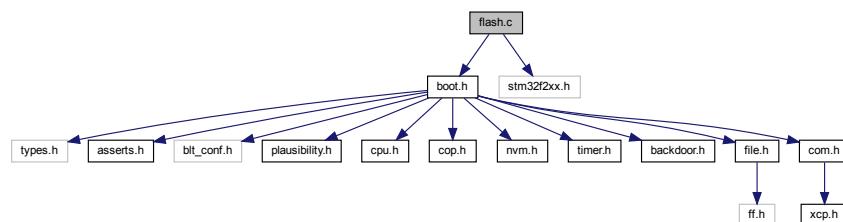
Referenced by [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.63 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32f2xx.h"

Include dependency graph for ARCMC3_STM32F2/flash.c:
```



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR** (0xff)
Value for an invalid flash sector.
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SECTORS** (sizeof(flashLayout)/sizeof(flashLayout[0]))
Total numbers of sectors in array flashLayout[].
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x184)
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- static **blt_bool FlashInitBlock** (**tFlashBlockInfo** *block, **blt_addr** address)
Copies data currently in flash to the block->data and sets the base address.
- static **tFlashBlockInfo** * **FlashSwitchBlock** (**tFlashBlockInfo** *block, **blt_addr** base_addr)
Switches blocks by programming the current one and initializing the next.
- static **blt_bool FlashAddToBlock** (**tFlashBlockInfo** *block, **blt_addr** address, **blt_int8u** *data, **blt_int32u** len)
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static **blt_bool FlashWriteBlock** (**tFlashBlockInfo** *block)
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static **blt_bool FlashEraseSectors** (**blt_int8u** first_sector, **blt_int8u** last_sector)
Erases the flash sectors from first_sector up until last_sector.
- static **blt_bool FlashEmptyCheckSector** (**blt_int8u** sector_num)
Checks if the flash sector is already completely erased.
- static **blt_int8u FlashGetSector** (**blt_addr** address)
Determines the flash sector the address is in.
- void **FlashInit** (**void**)
Initializes the flash driver.
- **blt_bool FlashWrite** (**blt_addr** addr, **blt_int32u** len, **blt_int8u** *data)
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- **blt_bool FlashErase** (**blt_addr** addr, **blt_int32u** len)
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- **blt_bool FlashWriteChecksum** (**void**)

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

- **blt_bool FlashVerifyChecksum (void)**
Verifies the checksum, which indicates that a valid user program is present and can be started.
- **blt_bool FlashDone (void)**
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- **blt_addr Flash GetUserProgBaseAddress (void)**
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const **tFlashSector flashLayout []**
If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to > 0 in `blt_conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.
- static **tFlashBlockInfo blockInfo**
Local variable with information about the flash block that is currently being operated on.
- static **tFlashBlockInfo bootBlockInfo**
Local variable with information about the flash boot block.

7.63.1 Detailed Description

Bootloader flash driver source file.

7.63.2 Function Documentation

7.63.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.63.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.63.2.3 FlashEmptyCheckSector()

```
static blt_bool FlashEmptyCheckSector (
    blt_int8u sector_num ) [static]
```

Checks if the flash sector is already completely erased.

Parameters

| | |
|-------------------------|--|
| <code>sector_num</code> | Sector number. Note that this is the sector_num element of the flashLayout array, not an index into the array. |
|-------------------------|--|

Returns

BLT_TRUE if the flash sector is already erased, BLT_FALSE otherwise.

Referenced by [FlashEraseSectors\(\)](#).

7.63.2.4 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.63.2.5 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector,
    blt_int8u last_sector ) [static]
```

Erases the flash sectors from *first_sector* up until *last_sector*.

Parameters

| | |
|---------------------|----------------------------|
| <i>first_sector</i> | First flash sector number. |
| <i>last_sector</i> | Last flash sector number. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.63.2.6 FlashGetSector()

```
static blt_int8u FlashGetSector (
    blt_addr address ) [static]
```

Determines the flash sector the *address* is in.

Parameters

| | |
|----------------|------------------------------|
| <i>address</i> | Address in the flash sector. |
|----------------|------------------------------|

Returns

Flash sector number or FLASH_INVALID_SECTOR.

Referenced by [FlashErase\(\)](#), and [FlashWrite\(\)](#).

7.63.2.7 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.63.2.8 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.63.2.9 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.63.2.10 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.63.2.11 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.63.2.12 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.63.2.13 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.63.2.14 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.63.3 Variable Documentation

7.63.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.63.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block needs to be written twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way it is possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.63.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

Initial value:

```
=  
{
```

```
{ 0x08008000, 0x04000, 2 },  
{ 0x0800c000, 0x04000, 3 },  
{ 0x08010000, 0x10000, 4 },  
{ 0x08020000, 0x20000, 5 },  
{ 0x08040000, 0x20000, 6 },  
{ 0x08060000, 0x20000, 7 },  
{ 0x08080000, 0x20000, 8 },  
{ 0x080A0000, 0x20000, 9 },  
{ 0x080C0000, 0x20000, 10 },  
{ 0x080E0000, 0x20000, 11 },  
}
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory.

Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

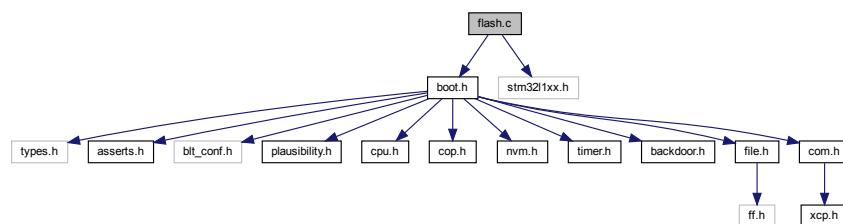
Referenced by [FlashEmptyCheckSector\(\)](#), [FlashEraseSectors\(\)](#), [FlashGetSector\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.64 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32l1xx.h"
```

Include dependency graph for ARMCM3_STM32L1/flash.c:



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SEGMENTS** (sizeof(**flashLayout**)/sizeof(**flashLayout**[0]))
*Total numbers of segments in array **flashLayout**.*
- #define **FLASH_LAST_SEGMENT_IDX** (**FLASH_TOTAL_SEGMENTS**-1)
*Index of the last segment in array **flashLayout**.*
- #define **FLASH_START_ADDRESS** (**flashLayout**[0].sector_start)
Start address of the bootloader programmable flash.
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **FLASH_ERASE_BLOCK_SIZE** (2048)
Number of bytes to erase per erase operation. Usually this is set to the value as defined by hardware (FLASH_→ PAGE_SIZE). However, on the STM32L1, this value is less than FLASH_WRITE_BLOCK_SIZE, which would lead to problems. Therefore this macro was set to a value that is a multiple of bothFLASH_WRITE_BLOCK_SIZE and FLASH_PAGE_SIZE.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x13C)
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- static `blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- static `tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- static `blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static `blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- void `FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- static `tFlashBlockInfo blockInfo`
Local variable with information about the flash block that is currently being operated on.
- static `tFlashBlockInfo bootBlockInfo`
Local variable with information about the flash boot block.

7.64.1 Detailed Description

Bootloader flash driver source file.

7.64.2 Function Documentation

7.64.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.64.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.64.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.64.2.4 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.64.2.5 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.64.2.6 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.64.2.7 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.64.2.8 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.64.2.9 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.64.2.10 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.64.2.11 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.64.3 Variable Documentation

7.64.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.64.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block needs to be written twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way it is possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.64.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory.

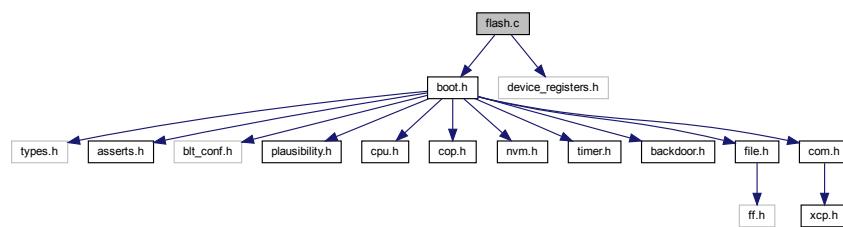
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten. The current flash layout does not reflect the minimum sector size of the physical flash (128 bytes), because this would make the table quite long and a waste of ROM. The minimum sector size is only really needed when erasing the flash. This can still be done in combination with macro FLASH_ERASE_BLOCK_SIZE.

Referenced by [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.65 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "device_registers.h"
Include dependency graph for ARMCM4_S32K14/flash.c:
```



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR_IDX** (0xff)
Value for an invalid sector entry index into flashLayout[].
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (1024)
Standard size of a flash block for writing.
- #define **FLASH_ERASE_BLOCK_SIZE** (FEATURE_FLS_PF_BLOCK_SECTOR_SIZE)
Standard size of a flash block for erasing. This is either 2 or 4 kb depending on the microcontroller derivative.
- #define **FLASH_TOTAL_SECTORS** (sizeof(flashLayout)/sizeof(flashLayout[0]))
Total numbers of sectors in array flashLayout[].
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **FLASH_FTFC_CMD_PROGRAM_PHRASE** (0x07U)
FTFC program phrase command code.
- #define **FLASH_FTFC_CMD_ERASE_SECTOR** (0x09U)
FTFC erase sector command code.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x3F8)
Offset into the user program's vector table where the checksum is located. For this target it is set to the second to last entry (254) in the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- static `blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- static `tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- static `blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static `blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static `blt_bool FlashEraseSectors (blt_int8u first_sector_idx, blt_int8u last_sector_idx)`
Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.
- static `blt_int8u FlashGetSectorIdx (blt_addr address)`
Determines the index into the flashLayout[] array of the flash sector that the specified address is in.
- static `START_FUNCTION_DECLARATION_RAMSECTION void FlashCommandSequence (void)`
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- void `FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static `tFlashBlockInfo blockInfo`
Local variable with information about the flash block that is currently being operated on.
- static `tFlashBlockInfo bootBlockInfo`
Local variable with information about the flash boot block.

7.65.1 Detailed Description

Bootloader flash driver source file.

7.65.2 Function Documentation

7.65.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.65.2.2 FlashCommandSequence()

```
static START_FUNCTION_DEFINITION_RAMSECTION void FlashCommandSequence (
    void ) [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Use the FTFC module to run the flash command sequence. It is assumed that that command and its necessary parameters were already written to the correct FTFC registers.

Array with the layout of the flash memory.

Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

Attention

This function needs to run from RAM. It is configured such that the C start-up code automatically copies it from ROM to RAM in function `init_data_bss()`, which is called by the reset handler.

Returns

None.

Referenced by [FlashEraseSectors\(\)](#), and [FlashWriteBlock\(\)](#).

7.65.2.3 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

`BLT_TRUE` if successful, `BLT_FALSE` otherwise.

7.65.2.4 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------------|------------------|
| <code>addr</code> | Start address. |
| <code>len</code> | Length in bytes. |

Returns

`BLT_TRUE` if successful, `BLT_FALSE` otherwise.

7.65.2.5 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector_idx,
    blt_int8u last_sector_idx ) [static]
```

Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.

Parameters

| | |
|-------------------------|---|
| <i>first_sector_idx</i> | First flash sector number index into flashLayout[]. |
| <i>last_sector_idx</i> | Last flash sector number index into flashLayout[]. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.65.2.6 FlashGetSectorIdx()

```
static blt_int8u FlashGetSectorIdx (
    blt_addr address ) [static]
```

Determines the index into the flashLayout[] array of the flash sector that the specified address is in.

Parameters

| | |
|----------------|------------------------------|
| <i>address</i> | Address in the flash sector. |
|----------------|------------------------------|

Returns

Flash sector index in flashLayout[] or FLASH_INVALID_SECTOR_IDX.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.65.2.7 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.65.2.8 FlashInit()

```
void FlashInit (
    void  )
```

Initializes the flash driver.

Returns

none.

7.65.2.9 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.65.2.10 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is now being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.65.2.11 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.65.2.12 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.65.2.13 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.65.2.14 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.65.3 Variable Documentation

7.65.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.65.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

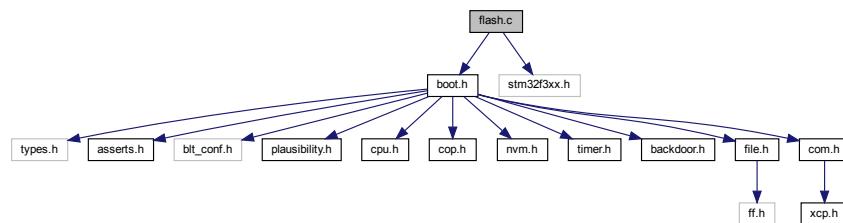
The first block of the user program holds the vector table, which on the STM32 is also the where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.66 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32f3xx.h"
Include dependency graph for ARMCM4_STM32F3/flash.c:
```



Data Structures

- struct [tFlashSector](#)
Flash sector descriptor type.
- struct [tFlashBlockInfo](#)
Structure type for grouping flash block information.

Macros

- `#define FLASH_INVALID_ADDRESS (0xffffffff)`
Value for an invalid flash address.
- `#define FLASH_WRITE_BLOCK_SIZE (512)`
Standard size of a flash block for writing.
- `#define FLASH_ERASE_SECTOR_SIZE (2048)`
Standard size of a flash sector for erasing.
- `#define FLASH_TOTAL_SEGMENTS (sizeof(flashLayout)/sizeof(flashLayout[0]))`
Total numbers of segments in array flashLayout[].
- `#define FLASH_LAST_SEGMENT_IDX (FLASH_TOTAL_SEGMENTS-1)`
Index of the last segment in array flashLayout[].
- `#define FLASH_START_ADDRESS (flashLayout[0].sector_start)`
Start address of the bootloader programmable flash.
- `#define FLASH_END_ADDRESS`
End address of the bootloader programmable flash.
- `#define BOOT_FLASH_VECTOR_TABLE_CS_OFFSET (0x188)`
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overriden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- `static blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- `static tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- `static blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- `static blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const [tFlashSector flashLayout \[\]](#)

If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to > 0 in `blt.conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.

- static [tFlashBlockInfo blockInfo](#)

Local variable with information about the flash block that is currently being operated on.

- static [tFlashBlockInfo bootBlockInfo](#)

Local variable with information about the flash boot block.

7.66.1 Detailed Description

Bootloader flash driver source file.

7.66.2 Function Documentation

7.66.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

`BLT_TRUE` if successful, `BLT_FALSE` otherwise.

Referenced by [FlashWrite\(\)](#).

7.66.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.66.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.66.2.4 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.66.2.5 FlashInit()

```
void FlashInit (
    void  )
```

Initializes the flash driver.

Returns

none.

7.66.2.6 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.66.2.7 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.66.2.8 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.66.2.9 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.66.2.10 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------------|--|
| <code>block</code> | Pointer to flash block info structure to operate on. |
|--------------------|--|

Returns

`BLT_TRUE` if successful, `BLT_FALSE` otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.66.2.11 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

`BLT_TRUE` if successful, `BLT_FALSE` otherwise.

7.66.3 Variable Documentation

7.66.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is `FLASH_WRITE_BLOCK_SIZE`. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.66.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also the where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.66.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array wit the layout of the flash memory.

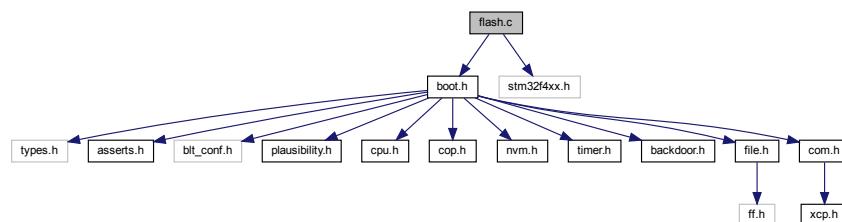
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

Referenced by [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.67 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32f4xx.h"
Include dependency graph for ARMCM4_STM32F4/flash.c:
```



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR** (0xff)
Value for an invalid flash sector.
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SECTORS** (sizeof(flashLayout)/sizeof(flashLayout[0]))
Total numbers of sectors in array flashLayout[].
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x188)
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- static **blt_bool FlashInitBlock** (**tFlashBlockInfo** *block, **blt_addr** address)
Copies data currently in flash to the block->data and sets the base address.
- static **tFlashBlockInfo** * **FlashSwitchBlock** (**tFlashBlockInfo** *block, **blt_addr** base_addr)
Switches blocks by programming the current one and initializing the next.
- static **blt_bool FlashAddToBlock** (**tFlashBlockInfo** *block, **blt_addr** address, **blt_int8u** *data, **blt_int32u** len)
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static **blt_bool FlashWriteBlock** (**tFlashBlockInfo** *block)
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static **blt_bool FlashEraseSectors** (**blt_int8u** first_sector, **blt_int8u** last_sector)
Erases the flash sectors from first_sector up until last_sector.
- static **blt_bool FlashEmptyCheckSector** (**blt_int8u** sector_num)
Checks if the flash sector is already completely erased.
- static **blt_int8u FlashGetSector** (**blt_addr** address)
Determines the flash sector the address is in.
- void **FlashInit** (**void**)
Initializes the flash driver.
- **blt_bool FlashWrite** (**blt_addr** addr, **blt_int32u** len, **blt_int8u** *data)
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- **blt_bool FlashErase** (**blt_addr** addr, **blt_int32u** len)
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- **blt_bool FlashWriteChecksum** (**void**)

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

- **blt_bool FlashVerifyChecksum (void)**
Verifies the checksum, which indicates that a valid user program is present and can be started.
- **blt_bool FlashDone (void)**
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- **blt_addr Flash GetUserProgBaseAddress (void)**
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const **tFlashSector flashLayout []**
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- static **tFlashBlockInfo blockInfo**
Local variable with information about the flash block that is currently being operated on.
- static **tFlashBlockInfo bootBlockInfo**
Local variable with information about the flash boot block.

7.67.1 Detailed Description

Bootloader flash driver source file.

7.67.2 Function Documentation

7.67.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.67.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.67.2.3 FlashEmptyCheckSector()

```
static blt_bool FlashEmptyCheckSector (
    blt_int8u sector_num ) [static]
```

Checks if the flash sector is already completely erased.

Parameters

| | |
|-------------------------|--|
| <code>sector_num</code> | Sector number. Note that this is the sector_num element of the flashLayout array, not an index into the array. |
|-------------------------|--|

Returns

BLT_TRUE if the flash sector is already erased, BLT_FALSE otherwise.

Referenced by [FlashEraseSectors\(\)](#).

7.67.2.4 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.67.2.5 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector,
    blt_int8u last_sector ) [static]
```

Erases the flash sectors from *first_sector* up until *last_sector*.

Parameters

| | |
|---------------------|----------------------------|
| <i>first_sector</i> | First flash sector number. |
| <i>last_sector</i> | Last flash sector number. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.67.2.6 FlashGetSector()

```
static blt_int8u FlashGetSector (
    blt_addr address ) [static]
```

Determines the flash sector the *address* is in.

Parameters

| | |
|----------------|------------------------------|
| <i>address</i> | Address in the flash sector. |
|----------------|------------------------------|

Returns

Flash sector number or FLASH_INVALID_SECTOR.

Referenced by [FlashErase\(\)](#), and [FlashWrite\(\)](#).

7.67.2.7 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.67.2.8 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.67.2.9 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.67.2.10 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.67.2.11 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.67.2.12 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.67.2.13 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.67.2.14 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.67.3 Variable Documentation

7.67.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.67.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way it is possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.67.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory.

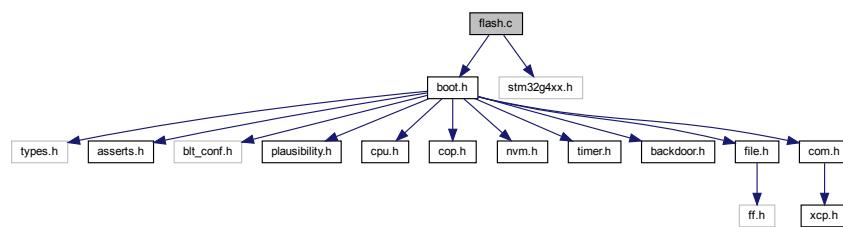
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

Referenced by [FlashEmptyCheckSector\(\)](#), [FlashEraseSectors\(\)](#), [FlashGetSector\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.68 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32g4xx.h"
Include dependency graph for ARMCM4_STM32G4/flash.c:
```



Data Structures

- struct [tFlashSector](#)
Flash sector descriptor type.
- struct [tFlashBlockInfo](#)
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR_IDX** (0xff)
Value for an invalid sector entry index into flashLayout[].
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_ERASE_PAGE_SIZE** ([FLASH_PAGE_SIZE](#))
Standard size of a flash page for erasing. note that a flash sector can have multiple pages.
- #define **FLASH_TOTAL_SECTORS** (sizeof([flashLayout](#))/sizeof([flashLayout\[0\]](#)))
Total numbers of sectors in array flashLayout[].
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x1D8)
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- static `blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- static `tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- static `blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static `blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static `blt_bool FlashEraseSectors (blt_int8u first_sector_idx, blt_int8u last_sector_idx)`
Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.
- static `blt_int8u FlashGetSectorIdx (blt_addr address)`
Determines the index into the flashLayout[] array of the flash sector that the specified address is in.
- static `blt_int32u FlashGetPage (blt_addr address)`
Determines the flash page number that the specified address belongs to.
- static `blt_int32u FlashGetBank (blt_addr address)`
Determines the flash bank number that the specified address belongs to.
- static `blt_bool FlashVerifyBankMode (void)`
Determines the flash banking mode is configured properly for this flash driver. This flash driver assumes that dual banking mode is configured, if the flash device supports dual banking mode. This is default mode as configured by ST in the option bytes.
- void `FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- static `tFlashBlockInfo blockInfo`
Local variable with information about the flash block that is currently being operated on.
- static `tFlashBlockInfo bootBlockInfo`
Local variable with information about the flash boot block.

7.68.1 Detailed Description

Bootloader flash driver source file.

7.68.2 Function Documentation

7.68.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.68.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.68.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.68.2.4 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector_idx,
    blt_int8u last_sector_idx ) [static]
```

Erases the flash sectors from indices *first_sector_idx* up until *last_sector_idx* into the *flashLayout[]* array.

Parameters

| | |
|-------------------------|---|
| <i>first_sector_idx</i> | First flash sector number index into <i>flashLayout[]</i> . |
| <i>last_sector_idx</i> | Last flash sector number index into <i>flashLayout[]</i> . |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.68.2.5 FlashGetBank()

```
static blt_int32u FlashGetBank (
    blt_addr address ) [static]
```

Determines the flash bank number that the specified address belongs to.

Parameters

| | |
|----------------------|-----------------------|
| <code>address</code> | Flash memory address. |
|----------------------|-----------------------|

Returns

The flash bank number that this address belongs to. It can be either FLASH_BANK_1 or FLASH_BANK_2.

Referenced by [FlashEraseSectors\(\)](#), and [FlashGetPage\(\)](#).

7.68.2.6 FlashGetPage()

```
static blt_int32u FlashGetPage (
    blt_addr address ) [static]
```

Determines the flash page number that the specified address belongs to.

Parameters

| | |
|----------------------|-----------------------|
| <code>address</code> | Flash memory address. |
|----------------------|-----------------------|

Returns

The flash page number that this address belongs to.

Referenced by [FlashEraseSectors\(\)](#).

7.68.2.7 FlashGetSectorIdx()

```
static blt_int8u FlashGetSectorIdx (
    blt_addr address ) [static]
```

Determines the index into the `flashLayout[]` array of the flash sector that the specified address is in.

Parameters

| | |
|----------------------|------------------------------|
| <code>address</code> | Address in the flash sector. |
|----------------------|------------------------------|

Returns

Flash sector index in `flashLayout[]` or `FLASH_INVALID_SECTOR_IDX`.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.68.2.8 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.68.2.9 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.68.2.10 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.68.2.11 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is now being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.68.2.12 FlashVerifyBankMode()

```
static blt_bool FlashVerifyBankMode (
    void ) [static]
```

Determines the flash banking mode is configured properly for this flash driver. This flash driver assumes that dual banking mode is configured, if the flash device supports dual banking mode. This is default mode as configured by ST in the option bytes.

Returns

BLT_TRUE if the flash banking mode is configured properly as supported by this flash driver, BLT_FALSE otherwise.

Referenced by [FlashEraseSectors\(\)](#), and [FlashWriteBlock\(\)](#).

7.68.2.13 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.68.2.14 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.68.2.15 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.68.2.16 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.68.3 Variable Documentation

7.68.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.68.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way it is possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.68.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory.

Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

Attention

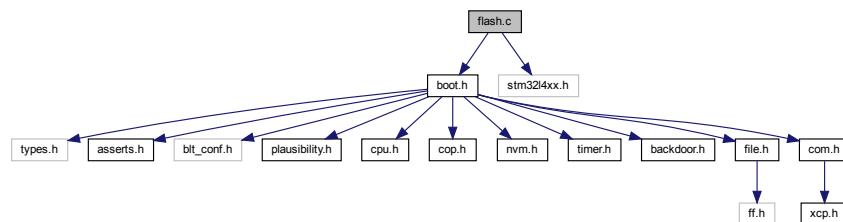
For category 3 flash device, which have dual banking mode support, this flash driver only supports dual banking mode configuration, not single banking mode. Simply because that is the default configuration that ST programs in the option bytes. This means that flash pages are always 2kb in size. Some sectors span multiple pages in this table. The only reason for this is to not make the table unnecessarily long, which would just waste flash space.

Referenced by [FlashEraseSectors\(\)](#), [FlashGetSectorIdx\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.69 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm3214xx.h"
Include dependency graph for ARMCM4_STM32L4/flash.c:
```



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_ERASE_SECTOR_SIZE** (2048)
Standard size of a flash sector for erasing.
- #define **FLASH_TOTAL_SEGMENTS** (sizeof(**flashLayout**)/sizeof(**flashLayout**[0]))
*Total numbers of segments in array **flashLayout**.*
- #define **FLASH_LAST_SEGMENT_IDX** (**FLASH_TOTAL_SEGMENTS**-1)
*Index of the last segment in array **flashLayout**.*
- #define **FLASH_START_ADDRESS** (**flashLayout**[0].sector_start)
Start address of the bootloader programmable flash.
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x188)
*Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in **blt_conf.h**, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.*

Functions

- static `blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- static `tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- static `blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static `blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static `blt_int32u FlashGetPage (blt_addr address)`
Gets the page number of the address relative to the bank.
- static `blt_int32u FlashGetBank (blt_addr address)`
Obtains the bank of the given address. The 1024kb version of the flash device contains 2 banks that can be swapped. This feature breaks the link between a bank number and flash addresses. This function obtains the bank number that is currently at the given address.
- void `FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased. Note that the term sector used by this flash driver is equivalent to the term page in the STM32L4x reference manual.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- static `tFlashBlockInfo blockInfo`
Local variable with information about the flash block that is currently being operated on.
- static `tFlashBlockInfo bootBlockInfo`
Local variable with information about the flash boot block.

7.69.1 Detailed Description

Bootloader flash driver source file.

7.69.2 Function Documentation

7.69.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.69.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.69.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased. Note that the term sector used by this flash driver is equivalent to the term page in the STM32L4x reference manual.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.69.2.4 FlashGetBank()

```
static blt_int32u FlashGetBank (
    blt_addr address ) [static]
```

Obtains the bank of the given address. The 1024kb version of the flash device contains 2 banks that can be swapped. This feature breaks the link between a bank number and flash addresses. This function obtains the bank number that is currently at the given address.

Parameters

| | |
|----------------|----------------------------|
| <i>address</i> | Address in the flash bank. |
|----------------|----------------------------|

Returns

The flash bank of the given address: FLASH_BANK_1 or FLASH_BANK_2.

Referenced by [FlashErase\(\)](#).

7.69.2.5 FlashGetPage()

```
static blt_int32u FlashGetPage (
    blt_addr address ) [static]
```

Gets the page number of the address relative to the bank.

Parameters

| | |
|----------------|----------------------------|
| <i>address</i> | Address in the flash bank. |
|----------------|----------------------------|

Returns

The page of the given address: 0..255.

Referenced by [FlashErase\(\)](#).

7.69.2.6 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.69.2.7 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.69.2.8 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.69.2.9 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.69.2.10 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.69.2.11 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.69.2.12 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.69.2.13 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.69.3 Variable Documentation

7.69.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.69.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way it is possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.69.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory. Note that the current flash driver supports the STM32L4x1, STM32L4x5 and STM32L4x6 derivatives in the STM32L4 family of microcontrollers.

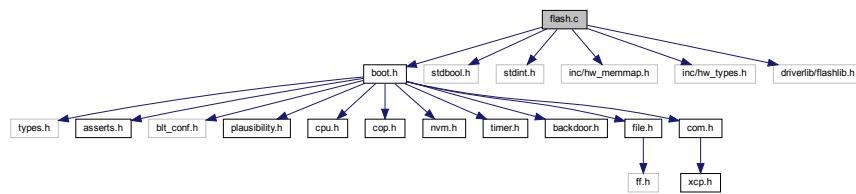
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

Referenced by [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.70 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/flashlib.h"
Include dependency graph for ARMCM4_TM4C/flash.c:
```



Data Structures

- struct [tFlashSector](#)
Flash sector descriptor type.
- struct [tFlashBlockInfo](#)
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR** (0xff)
Value for an invalid flash sector.
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (1024)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SECTORS** (sizeof([flashLayout](#))/sizeof([flashLayout](#)[0]))
Total numbers of sectors in array [flashLayout](#)[].
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **FLASH_ERASE_BLOCK_SIZE** (0x400)
Number of bytes to erase per erase operation.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x26C)
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in [blt_conf.h](#), because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- static `blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- static `tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- static `blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static `blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static `blt_bool FlashEraseSectors (blt_int8u first_sector, blt_int8u last_sector)`
Erases the flash sectors from first_sector up until last_sector.
- static `blt_int8u FlashGetSector (blt_addr address)`
Determines the flash sector the address is in.
- static `blt_addr FlashGetSectorBaseAddr (blt_int8u sector)`
Determines the flash sector base address.
- static `blt_addr FlashGetSectorSize (blt_int8u sector)`
Determines the flash sector size.
- void `FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- static `tFlashBlockInfo blockInfo`
Local variable with information about the flash block that is currently being operated on.
- static `tFlashBlockInfo bootBlockInfo`
Local variable with information about the flash boot block.

7.70.1 Detailed Description

Bootloader flash driver source file.

7.70.2 Function Documentation

7.70.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.70.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.70.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.70.2.4 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector,
    blt_int8u last_sector ) [static]
```

Erases the flash sectors from *first_sector* up until *last_sector*.

Parameters

| | |
|---------------------|----------------------------|
| <i>first_sector</i> | First flash sector number. |
| <i>last_sector</i> | Last flash sector number. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.70.2.5 FlashGetSector()

```
static blt_int8u FlashGetSector (
    blt_addr address ) [static]
```

Determines the flash sector the address is in.

Parameters

| | |
|----------------------|------------------------------|
| <code>address</code> | Address in the flash sector. |
|----------------------|------------------------------|

Returns

Flash sector number or FLASH_INVALID_SECTOR.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.70.2.6 FlashGetSectorBaseAddr()

```
static blt_addr FlashGetSectorBaseAddr (
    blt_int8u sector ) [static]
```

Determines the flash sector base address.

Parameters

| | |
|---------------------|------------------------------------|
| <code>sector</code> | Sector to get the base address of. |
|---------------------|------------------------------------|

Returns

Flash sector base address or FLASH_INVALID_ADDRESS.

Referenced by [FlashEraseSectors\(\)](#).

7.70.2.7 FlashGetSectorSize()

```
static blt_addr FlashGetSectorSize (
    blt_int8u sector ) [static]
```

Determines the flash sector size.

Parameters

| | |
|---------------------|----------------------------|
| <code>sector</code> | Sector to get the size of. |
|---------------------|----------------------------|

Returns

Flash sector size or 0.

Referenced by [FlashEraseSectors\(\)](#).

7.70.2.8 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.70.2.9 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.70.2.10 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.70.2.11 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.70.2.12 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.70.2.13 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.70.2.14 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.70.2.15 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.70.3 Variable Documentation

7.70.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.70.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way it is possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.70.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory.

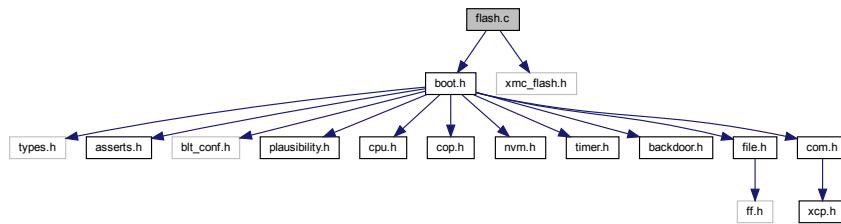
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten. The current flash layout does not reflect the minimum sector size of the physical flash (1 - 2kb), because this would make the table quite long and a waste of ROM. The minimum sector size is only really needed when erasing the flash. This can still be done in combination with macro FLASH_ERASE_BLOCK_SIZE.

Referenced by [FlashEraseSectors\(\)](#), [FlashGetSector\(\)](#), [FlashGetSectorBaseAddr\(\)](#), [FlashGetSectorSize\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.71 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "xmc_flash.h"
Include dependency graph for ARMCM4_XMC4/flash.c:
```



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR** (0xff)
Value for an invalid flash sector.
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (1024)
Standard size of a flash block for writing. It should be large enough so that the OpenBLT checksum fits in the first (boot) block.
- #define **FLASH_TOTAL_SECTORS** (sizeof(**flashLayout**)/sizeof(**flashLayout**[0]))
*Total numbers of sectors in array **flashLayout**[].*
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x200)
*Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in **blt_conf.h**, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.*
- #define **FLASH_WRITE_PAGE_SIZE** (256)
Minimum amount of bytes that can be programmed to flash at a time. It is hardware dependent.
- #define **FLASH_UNCACHED_BASE_ADDR** (0x0C000000U)
Base address in the memory map for uncached flash. It is hardware dependent.
- #define **FLASH_CACHED_BASE_ADDR** (0x08000000U)
Base address in the memory map for cached flash. It is hardware dependent.
- #define **FLASH_ERASE_TIME_MAX_MS** (6600)
Maximum time for a sector erase operation as specified by the XCM4xxx data-sheet with an added 20% margin.
- #define **FLASH_PROGRAM_TIME_MAX_MS** (13)
Maximum time for a page program operation as specified by the XCM4xxx data-sheet with an added 20% margin.

Functions

- static `blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- static `tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- static `blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static `blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static `blt_bool FlashEraseSectors (blt_int8u first_sector, blt_int8u last_sector)`
Erases the flash sectors from first_sector up until last_sector.
- static `blt_bool FlashEmptyCheckSector (blt_int8u sector_num)`
Checks if the flash sector is already completely erased.
- static `blt_int8u FlashGetSector (blt_addr address)`
Determines the flash sector the address is in.
- static `blt_addr FlashGetSectorBaseAddr (blt_int8u sector)`
Obtains the base address of the specified sector.
- static `blt_addr FlashTranslateToNonCachedAddress (blt_addr address)`
The XMC4xxx has its PFLASH accessible in the memory map in two regions. One is the non-cached region starting at FLASH_UNCACHED_BASE_ADDR and the other is the cached region starting at FLASH_CACHED_BASE_ADDR. Flash erase and programming operations need to operate on addresses in the non-cached region. It is possible that the caller of this driver's API functions, specifies memory addresses in the cached region. This function automatically translates the memory address from cached to non-cached.
- void `FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- static `tFlashBlockInfo blockInfo`
Local variable with information about the flash block that is currently being operated on.
- static `tFlashBlockInfo bootBlockInfo`
Local variable with information about the flash boot block.

7.71.1 Detailed Description

Bootloader flash driver source file.

7.71.2 Function Documentation

7.71.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.71.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.71.2.3 FlashEmptyCheckSector()

```
static blt_bool FlashEmptyCheckSector (
    blt_int8u sector_num ) [static]
```

Checks if the flash sector is already completely erased.

Parameters

| | |
|-------------------|--|
| <i>sector_num</i> | Sector number. Note that this is the sector_num element of the flashLayout array, not an index into the array. |
|-------------------|--|

Returns

BLT_TRUE if the flash sector is already erased, BLT_FALSE otherwise.

Referenced by [FlashEraseSectors\(\)](#).

7.71.2.4 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.71.2.5 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector,
    blt_int8u last_sector ) [static]
```

Erases the flash sectors from *first_sector* up until *last_sector*.

Parameters

| | |
|---------------------|----------------------------|
| <i>first_sector</i> | First flash sector number. |
| <i>last_sector</i> | Last flash sector number. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.71.2.6 FlashGetSector()

```
static blt_int8u FlashGetSector (
    blt_addr address ) [static]
```

Determines the flash sector the address is in.

Parameters

| | |
|----------------|------------------------------|
| <i>address</i> | Address in the flash sector. |
|----------------|------------------------------|

Returns

Flash sector number or FLASH_INVALID_SECTOR.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.71.2.7 FlashGetSectorBaseAddr()

```
static blt_addr FlashGetSectorBaseAddr (
    blt_int8u sector ) [static]
```

Obtains the base address of the specified sector.

Parameters

| | |
|---------------|------------------------------------|
| <i>sector</i> | Sector to get the base address of. |
|---------------|------------------------------------|

Returns

Base Base address of the sector if found, FLASH_INVALID_ADDRESS otherwise.

Referenced by [FlashEraseSectors\(\)](#).

7.71.2.8 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.71.2.9 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.71.2.10 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.71.2.11 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.71.2.12 FlashTranslateToNonCachedAddress()

```
static blt_addr FlashTranslateToNonCachedAddress (
    blt_addr address ) [static]
```

The XMC4xxx has its PFLASH accessible in the memory map in two regions. One is the non-cached region starting at FLASH_UNCACHED_BASE_ADDR and the other is the cached region starting at FLASH_CACHED_BASE_ADDR. Flash erase and programming operations need to operate on addresses in the non-cached region. It is possible that the caller of this driver's API functions, specifies memory addresses in the cached region. This function automatically translates the memory address from cached to non-cached.

Parameters

| | |
|----------------|-----------------------|
| <i>address</i> | Address to translate. |
|----------------|-----------------------|

Returns

Translated address.

Referenced by [FlashErase\(\)](#), and [FlashWrite\(\)](#).

7.71.2.13 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.71.2.14 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.71.2.15 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.71.2.16 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.71.3 Variable Documentation

7.71.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.71.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.71.3.3 flashLayout

```
const tFlashSector flashLayout[ ] [static]
```

Initial value:

```
=
{

{ 0x0c00c000, 0x04000, 3 },
{ 0x0c010000, 0x04000, 4 },
{ 0x0c014000, 0x04000, 5 },
{ 0x0c018000, 0x04000, 6 },
{ 0x0c01c000, 0x04000, 7 },
{ 0x0c020000, 0x20000, 8 },
{ 0x0c040000, 0x40000, 9 },
{ 0x0c080000, 0x40000, 10 },
{ 0x0c0C0000, 0x40000, 11 },
}
```

If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to > 0 in `blt_conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.

Array wit the layout of the flash memory.

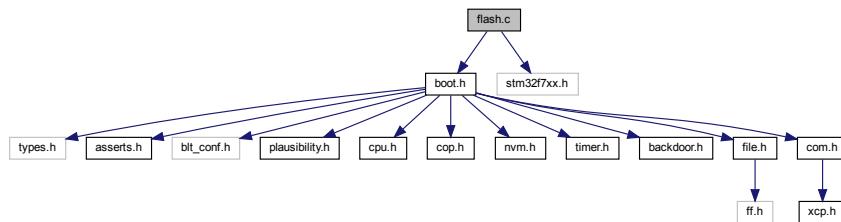
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten. Note that the table contains uncached addresses, because flash program/ erase operations need to be performed on uncached addresses. This flash driver automatically translated cached to uncached addresses, so there is no need for the user to adjust this when calling this driver's API.

Referenced by [FlashEmptyCheckSector\(\)](#), [FlashEraseSectors\(\)](#), [FlashGetSector\(\)](#), [FlashGetSectorBaseAddr\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.72 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32f7xx.h"
Include dependency graph for ARMCM7_STM32F7/flash.c:
```



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR** (0xff)
Value for an invalid flash sector.
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SECTORS** (sizeof(flashLayout)/sizeof(flashLayout[0]))
Total numbers of sectors in array flashLayout[].
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x1C8)
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- static **blt_bool FlashInitBlock** (**tFlashBlockInfo** *block, **blt_addr** address)
Copies data currently in flash to the block->data and sets the base address.
- static **tFlashBlockInfo** * **FlashSwitchBlock** (**tFlashBlockInfo** *block, **blt_addr** base_addr)
Switches blocks by programming the current one and initializing the next.
- static **blt_bool FlashAddToBlock** (**tFlashBlockInfo** *block, **blt_addr** address, **blt_int8u** *data, **blt_int32u** len)
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static **blt_bool FlashWriteBlock** (**tFlashBlockInfo** *block)
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static **blt_bool FlashEraseSectors** (**blt_int8u** first_sector, **blt_int8u** last_sector)
Erases the flash sectors from first_sector up until last_sector.
- static **blt_bool FlashEmptyCheckSector** (**blt_int8u** sector_num)
Checks if the flash sector is already completely erased.
- static **blt_int8u FlashGetSector** (**blt_addr** address)
Determines the flash sector the address is in.
- static **blt_bool FlashIsSingleBankMode** (**void**)
Determines the flash is configured in single bank mode, which is required by this flash driver.
- void **FlashInit** (**void**)
Initializes the flash driver.
- **blt_bool FlashWrite** (**blt_addr** addr, **blt_int32u** len, **blt_int8u** *data)
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- **blt_bool FlashErase** (**blt_addr** addr, **blt_int32u** len)

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

- `blt_bool FlashWriteChecksum (void)`

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

- `blt_bool FlashVerifyChecksum (void)`

Verifies the checksum, which indicates that a valid user program is present and can be started.

- `blt_bool FlashDone (void)`

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

- `blt_addr Flash GetUserProgBaseAddress (void)`

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`

If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to > 0 in `blt_conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.

- static `tFlashBlockInfo blockInfo`

Local variable with information about the flash block that is currently being operated on.

- static `tFlashBlockInfo bootBlockInfo`

Local variable with information about the flash boot block.

7.72.1 Detailed Description

Bootloader flash driver source file.

7.72.2 Function Documentation

7.72.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.72.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.72.2.3 FlashEmptyCheckSector()

```
static blt_bool FlashEmptyCheckSector (
    blt_int8u sector_num ) [static]
```

Checks if the flash sector is already completely erased.

Parameters

| | |
|-------------------|--|
| <i>sector_num</i> | Sector number. Note that this is the sector_num element of the flashLayout array, not an index into the array. |
|-------------------|--|

Returns

BLT_TRUE if the flash sector is already erased, BLT_FALSE otherwise.

Referenced by [FlashEraseSectors\(\)](#).

7.72.2.4 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.72.2.5 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector,
    blt_int8u last_sector ) [static]
```

Erases the flash sectors from *first_sector* up until *last_sector*.

Parameters

| | |
|---------------------|----------------------------|
| <i>first_sector</i> | First flash sector number. |
| <i>last_sector</i> | Last flash sector number. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.72.2.6 FlashGetSector()

```
static blt_int8u FlashGetSector (
    blt_addr address ) [static]
```

Determines the flash sector the address is in.

Parameters

| | |
|----------------------|------------------------------|
| <code>address</code> | Address in the flash sector. |
|----------------------|------------------------------|

Returns

Flash sector number or FLASH_INVALID_SECTOR.

Referenced by [FlashErase\(\)](#), and [FlashWrite\(\)](#).

7.72.2.7 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.72.2.8 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.72.2.9 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.72.2.10 FlashIsSingleBankMode()

```
static blt_bool FlashIsSingleBankMode (
    void ) [static]
```

Determines the flash is configured in single bank mode, which is required by this flash driver.

Returns

BLT_TRUE if the flash is in single bank mode, BLT_FALSE otherwise.

Referenced by [FlashEraseSectors\(\)](#), and [FlashWriteBlock\(\)](#).

7.72.2.11 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.72.2.12 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.72.2.13 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.72.2.14 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.72.2.15 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.72.3 Variable Documentation

7.72.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.72.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way it is possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.72.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

Initial value:

```
=
{
    { 0x08008000, 0x08000, 1},
    { 0x08010000, 0x08000, 2},
    { 0x08018000, 0x08000, 3},
    { 0x08020000, 0x20000, 4},
    { 0x08040000, 0x40000, 5},
    { 0x08080000, 0x40000, 6},
    { 0x080C0000, 0x40000, 7},
}
```

If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to > 0 in `blt_conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.

Array wit the layout of the flash memory.

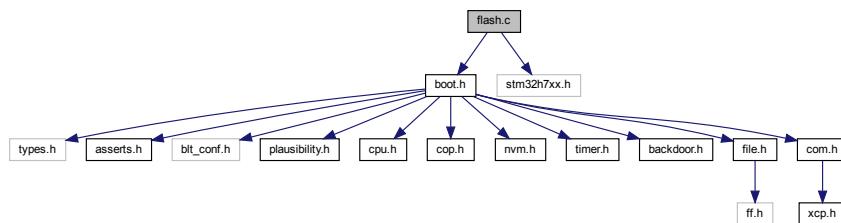
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

Referenced by [FlashEmptyCheckSector\(\)](#), [FlashEraseSectors\(\)](#), [FlashGetSector\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.73 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "stm32h7xx.h"
Include dependency graph for ARMCM7_STM32H7/flash.c:
```



Data Structures

- struct [tFlashSector](#)
Flash sector descriptor type.
- struct [tFlashBlockInfo](#)
Structure type for grouping flash block information.

Macros

- `#define FLASH_INVALID_SECTOR_IDX (0xff)`
Value for an invalid sector entry index into flashLayout[].
- `#define FLASH_INVALID_ADDRESS (0xffffffff)`
Value for an invalid flash address.
- `#define FLASH_WRITE_BLOCK_SIZE (1024)`
Standard size of a flash block for writing.
- `#define FLASH_TOTAL_SECTORS (sizeof(flashLayout)/sizeof(flashLayout[0]))`
Total numbers of sectors in array flashLayout[].
- `#define FLASH_END_ADDRESS`
End address of the bootloader programmable flash.
- `#define BOOT_FLASH_VECTOR_TABLE_CS_OFFSET (0x298)`
Offset into the user program's vector table where the checksum is located. For this target it is set to the end of the vector table. Note that the value can be overridden in blt_conf.h, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.

Functions

- `static blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- `static tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- `static blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- `static blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- `static blt_bool FlashEmptyCheckSector (blt_int8u sector_idx)`
Checks if the flash sector is already completely erased.
- `static blt_bool FlashEraseSectors (blt_int8u first_sector_idx, blt_int8u last_sector_idx)`
Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.
- `static blt_int8u FlashGetSectorIdx (blt_addr address)`
Determines the index into the flashLayout[] array of the flash sector that the specified address is in.
- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`

If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to > 0 in `blt.conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.

- static `tFlashBlockInfo blockInfo`

Local variable with information about the flash block that is currently being operated on.

- static `tFlashBlockInfo bootBlockInfo`

Local variable with information about the flash boot block.

7.73.1 Detailed Description

Bootloader flash driver source file.

7.73.2 Function Documentation

7.73.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------------|--|
| <code>block</code> | Pointer to flash block info structure to operate on. |
| <code>address</code> | Flash destination address. |
| <code>data</code> | Pointer to the byte array with data. |
| <code>len</code> | Number of bytes to add to the block. |

Returns

`BLT_TRUE` if successful, `BLT_FALSE` otherwise.

Referenced by [FlashWrite\(\)](#).

7.73.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.73.2.3 FlashEmptyCheckSector()

```
static blt_bool FlashEmptyCheckSector (
    blt_int8u sector_idx ) [static]
```

Checks if the flash sector is already completely erased.

Parameters

| | |
|-------------------|---|
| <i>sector_idx</i> | flash sector number index into flashLayout[]. |
|-------------------|---|

Returns

BLT_TRUE if the flash sector is already erased, BLT_FALSE otherwise.

Referenced by [FlashEraseSectors\(\)](#).

7.73.2.4 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.73.2.5 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector_idx,
    blt_int8u last_sector_idx ) [static]
```

Erases the flash sectors from indices `first_sector_idx` up until `last_sector_idx` into the `flashLayout[]` array.

Parameters

| | |
|-------------------------------|---|
| <code>first_sector_idx</code> | First flash sector number index into <code>flashLayout[]</code> . |
| <code>last_sector_idx</code> | Last flash sector number index into <code>flashLayout[]</code> . |

Returns

`BLT_TRUE` if successful, `BLT_FALSE` otherwise.

Referenced by [FlashErase\(\)](#).

7.73.2.6 FlashGetSectorIdx()

```
static blt_int8u FlashGetSectorIdx (
    blt_addr address ) [static]
```

Determines the index into the `flashLayout[]` array of the flash sector that the specified address is in.

Parameters

| | |
|----------------------|------------------------------|
| <code>address</code> | Address in the flash sector. |
|----------------------|------------------------------|

Returns

Flash sector index in `flashLayout[]` or `FLASH_INVALID_SECTOR_IDX`.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.73.2.7 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the `flashLayout` table.

Returns

Base address.

7.73.2.8 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.73.2.9 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.73.2.10 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is now being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.73.2.11 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.73.2.12 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.73.2.13 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.73.2.14 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.73.3 Variable Documentation

7.73.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.73.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also the where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.73.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array wit the layout of the flash memory.

Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

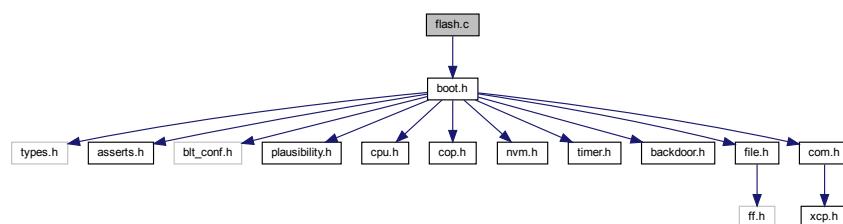
Referenced by [FlashEmptyCheckSector\(\)](#), [FlashEraseSectors\(\)](#), [FlashGetSectorIdx\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.74 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
```

Include dependency graph for HCS12/flash.c:



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.
- struct **tFlashRegs**
Structure type for the flash control registers.

Macros

- #define **FLASH_INVALID_SECTOR_IDX** (0xff)
Value for an invalid flash sector.
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SECTORS** (sizeof(**flashLayout**)/sizeof(**flashLayout[0]**))
*Total numbers of sectors in array **flashLayout[]**.*
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x82)
*Offset into the user program's vector table where the checksum is located. Note that the value can be overridden in **blt_conf.h**, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.*
- #define **FLASH_VECTOR_TABLE_SIZE** (0x80)
Total size of the vector table, excluding the bootloader specific checksum.
- #define **FLASH_START_ADDRESS** (**flashLayout[0].sector_start**)
Start address of the bootloader programmable flash.
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **FLASH_PAGE_SIZE** (0x4000) /* flash page size in bytes */
Size of a flash page on the HCS12.
- #define **FLASH_PAGE_OFFSET** (0x8000) /* physical start addr. of pages */
Physical start address of the HCS12 page window.
- #define **FLASH_PPAGE_REG** (*(volatile **blt_int8u** *) (0x0030))
PPAGE register to select a specific flash page.
- #define **FLASH_REGS_BASE_ADDRESS** (0x0100)
Base address of the flash related control registers.
- #define **FLASH** ((volatile **tFlashRegs** *) **FLASH_REGS_BASE_ADDRESS**)
Macro for accessing the flash related control registers.
- #define **FLASH_PROGRAM_WORD_CMD** (0x20)
Program word flash command.
- #define **FLASH_ERASE_SECTOR_CMD** (0x40)
Erase sector flash command.
- #define **FLASH_PAGES_PER_BLOCK** (8)
Number of flash pages in a block.
- #define **FLASH_BLOCK_SEL_MASK** (0x03)
Bitmask for selecting a block with flash pages.
- #define **PRDIV8_BIT** (0x40)
FCLKDIV - enable prescaler by 8 bit.
- #define **ACCERR_BIT** (0x10)
FSTAT - flash access error bit.

- #define **PVIOL_BIT** (0x20)
FSTAT - protection violation bit.
- #define **CBEIF_BIT** (0x80)
FSTAT - command buffer empty flag bit.
- #define **CBEIE_BIT** (0x80)
FCNFG - command buf. empty irq enable bit.
- #define **CCIE_BIT** (0x40)
FCNFG - command complete irq enable bit.
- #define **KEYACC_BIT** (0x20)
FCNFG - enable security key writing bit.

Typedefs

- typedef void(* **pFlashExeCmdFct**) (void)
Pointer type to flash command execution function.

Functions

- static **blt_bool FlashInitBlock** (**tFlashBlockInfo** *block, **blt_addr** address)
Copies data currently in flash to the block->data and sets the base address.
- static **tFlashBlockInfo** * **FlashSwitchBlock** (**tFlashBlockInfo** *block, **blt_addr** base_addr)
Switches blocks by programming the current one and initializing the next.
- static **blt_bool FlashAddToBlock** (**tFlashBlockInfo** *block, **blt_addr** address, **blt_int8u** *data, **blt_int32u** len)
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static **blt_bool FlashWriteBlock** (**tFlashBlockInfo** *block)
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static **blt_int8u FlashGetLinearAddrByte** (**blt_addr** addr)
Reads the byte value from the linear address.
- static **blt_int8u FlashGetPhysPage** (**blt_addr** addr)
Extracts the physical flash page number from a linear address.
- static **blt_int16u FlashGetPhysAddr** (**blt_addr** addr)
Extracts the physical address on the flash page number from a linear address.
- static void **FlashExecuteCommand** (void)
Executes the command. The actual code for the command execution is stored as location independant machine code in array flashExecCmd[]. The contents of this array are temporarily copied to RAM. This way the function can be executed from RAM avoiding problem when try to perform a flash operation on the same flash block that this driver is located.
- static **blt_bool FlashOperate** (**blt_int8u** cmd, **blt_addr** addr, **blt_int16u** data)
Prepares the flash command and executes it.
- void **FlashInit** (void)
Initializes the flash driver.
- **blt_bool FlashWrite** (**blt_addr** addr, **blt_int32u** len, **blt_int8u** *data)
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- **blt_bool FlashErase** (**blt_addr** addr, **blt_int32u** len)
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- **blt_bool FlashWriteChecksum** (void)

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

- **blt_bool FlashVerifyChecksum (void)**

Verifies the checksum, which indicates that a valid user program is present and can be started.

- **blt_bool FlashDone (void)**

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

- **blt_addr Flash GetUserProgBaseAddress (void)**

Obtains the base address of the flash memory available to the user program. This is basically the last address in the flashLayout table converted to the physical address on the last page (0x3f), because this is where the address will be in.

Variables

- static const **tFlashSector flashLayout []**

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

- static const **blt_int8u flashExecCmd []**

Array with executable code for performing flash operations.

- static **tFlashBlockInfo blockInfo**

Local variable with information about the flash block that is currently being operated on.

- static **tFlashBlockInfo bootBlockInfo**

Local variable with information about the flash boot block.

- static **blt_int8u flashExecCmdRam [(sizeof(flashExecCmd)/sizeof(flashExecCmd[0]))]**

RAM buffer where the executable flash operation code is copied to.

- static **blt_int8u flashMaxNrBlocks**

Maximum number of supported blocks, which is determined dynamically to have code that is independent of the used HCS12 derivative.

7.74.1 Detailed Description

Bootloader flash driver source file.

7.74.2 Function Documentation

7.74.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.74.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.74.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.74.2.4 FlashExecuteCommand()

```
static void FlashExecuteCommand (
    void ) [static]
```

Executes the command. The actual code for the command execution is stored as location independant machine code in array `flashExecCmd[]`. The contents of this array are temporarily copied to RAM. This way the function can be executed from RAM avoiding problem when try to perform a flash operation on the same flash block that this driver is located.

Returns

none.

Referenced by [FlashOperate\(\)](#).

7.74.2.5 FlashGetLinearAddrByte()

```
static blt_int8u FlashGetLinearAddrByte (
    blt_addr addr ) [static]
```

Reads the byte value from the linear address.

Parameters

| | |
|-------------------|-----------------|
| <code>addr</code> | Linear address. |
|-------------------|-----------------|

Returns

The byte value located at the linear address.

Referenced by [FlashWriteBlock\(\)](#).

7.74.2.6 FlashGetPhysAddr()

```
static blt_int16u FlashGetPhysAddr (
    blt_addr addr ) [static]
```

Extracts the physical address on the flash page number from a linear address.

Parameters

| | |
|-------------------|-----------------|
| <code>addr</code> | Linear address. |
|-------------------|-----------------|

Returns

The physical address.

Referenced by [FlashGetLinearAddrByte\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashInitBlock\(\)](#), and [FlashOperate\(\)](#).

7.74.2.7 FlashGetPhysPage()

```
static blt_int8u FlashGetPhysPage (
    blt_addr addr ) [static]
```

Extracts the physical flash page number from a linear address.

Parameters

| | |
|-------------|-----------------|
| <i>addr</i> | Linear address. |
|-------------|-----------------|

Returns

The page number.

Referenced by [FlashGetLinearAddrByte\(\)](#), [FlashInitBlock\(\)](#), and [FlashOperate\(\)](#).

7.74.2.8 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the last address in the flashLayout table converted to the physical address on the last page (0x3f), because this is where the address will be in.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.74.2.9 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.74.2.10 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#).

7.74.2.11 FlashOperate()

```
static blt_bool FlashOperate (
    blt_int8u cmd,
    blt_addr addr,
    blt_int16u data ) [static]
```

Prepares the flash command and executes it.

Parameters

| | |
|-------------|--------------------------------------|
| <i>cmd</i> | Command to be launched. |
| <i>addr</i> | Physical address for operation. |
| <i>data</i> | Data to write to addr for operation. |

Returns

BLT_TRUE if operation was successful, otherwise BLT_FALSE.

Referenced by [FlashWriteBlock\(\)](#).

7.74.2.12 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.74.2.13 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.74.2.14 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.74.2.15 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#).

7.74.2.16 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.74.3 Variable Documentation

7.74.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), and [FlashInit\(\)](#).

7.74.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also the where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.74.3.3 flashExecCmd

```
const blt_int8u flashExecCmd[] [static]
```

Initial value:

```
=
{
    0x36,
    0x34,
    0xce, 0x01, 0x00,
    0x1a, 0x05,
    0x86, 0x80,
    0x6a, 0x00,
    0xa7, 0xa7, 0xa7, 0xa7,
    0x0f, 0x00, 0x40, 0xfc,
    0x30,
    0x32,
    0x3d
}
```

Array with executable code for performing flash operations.

This array contains the machine code to perform the actual command on the flash device, such as program or erase. the code is compiler and location independent. This allows us to copy it to a ram buffer and execute the code from ram. This way the flash driver can be located in flash memory without running into problems when erasing/programming the same flash block that contains the flash driver. the source code for the machine code is as follows: // launch the command FLASH->fstat = CBEIF_BIT; // wait at least 4 cycles (per AN2720) asm("nop"); asm("nop"); asm("nop"); asm("nop"); // wait for command to complete while ((FLASH->fstat & CCIF_BIT) != CCIF←_BIT);

Referenced by [FlashDone\(\)](#), and [FlashExecuteCommand\(\)](#).

7.74.3.4 flashLayout

```
const tFlashSector flashLayout[ ] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array wit the layout of the flash memory.

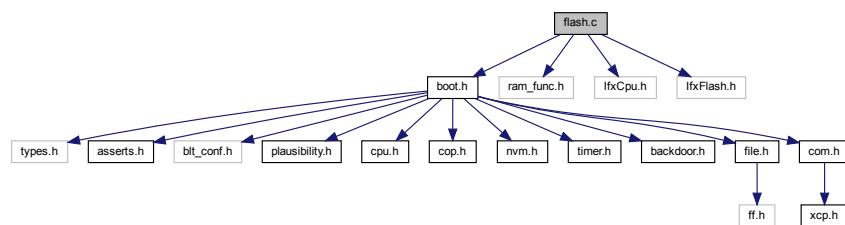
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten. This layout uses linear addresses only. For example, the first address on page 0x3F is: 0x3F * 0x4000 (page size) = 0xFC000. Note that page 0x3F is where the bootloader also resides and it has been entered as 8 chunks of 2kb. This allows flexibility for reserving more/less space for the bootloader in case its size changes in the future.

Referenced by [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.75 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "ram_func.h"
#include "IfxCpu.h"
#include "IfxFlash.h"
Include dependency graph for TRICORE_TC2/flash.c:
```



Data Structures

- struct [tFlashSector](#)
Flash sector descriptor type.
- struct [tFlashBlockInfo](#)
Structure type for grouping flash block information.

Macros

- `#define FLASH_INVALID_SECTOR_IDX (0xff)`
Value for an invalid sector entry index into flashLayout[].
- `#define FLASH_INVALID_ADDRESS (0xffffffff)`
Value for an invalid flash address.
- `#define FLASH_WRITE_BLOCK_SIZE (512)`
Standard size of a flash block for writing.
- `#define FLASH_TOTAL_SECTORS (sizeof(flashLayout)/sizeof(flashLayout[0]))`
Total numbers of sectors in array flashLayout[].
- `#define FLASH_END_ADDRESS`
End address of the bootloader programmable flash.
- `#define BOOT_FLASH_VECTOR_TABLE_CS_OFFSET (0x3C)`
Offset into the user program where the checksum is located. For this target it is set to the last 32-bits of the 64 byte (0x40) section at the start of the user program. The first 32 bytes (0x00..0x1F) are reserved for the BMHD table. The following 32 bytes (0x20..0x3F) is meant for the reset handler. The reset handler doesn't need the full 32 bytes that's reserved for it. Therefore the last 32-bit (0x3C..0x3F) can be used for storing the bootloader's signature checksum placeholder. Note that this macro value can be overriden in blt_conf.h, in case you want to reserve space for the signature checksum at a different memory location. Just make sure it is located in the first FLASH_WRITE_BLOCK_SIZE bytes of the user program. When changing this value, don't forget to update the location where you reserve space for the signature checksum in the user program accordingly. Otherwise the bootloader might overwrite important program code with the calculated signature checksum value, which can result in your user program not running properly.
- `#define FLASH_UNCACHED_BASE_ADDR (0xa0000000UL)`
Base address in the memory map for uncached flash. It is hardware dependent.
- `#define FLASH_CACHED_BASE_ADDR (0x80000000UL)`
Base address in the memory map for cached flash. It is hardware dependent.

Functions

- `static blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- `static tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- `static blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- `static blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- `static blt_bool FlashEraseSectors (blt_int8u first_sector_idx, blt_int8u last_sector_idx)`
Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.
- `static blt_bool FlashEraseLogicalSector (blt_addr log_sector_base_addr)`
Erases one logical sector starting at the specified base address.
- `static blt_bool FlashWritePage (blt_addr page_base_addr, blt_int8u const *page_data)`
Programs data to a flash page starting at the specified base address.
- `static blt_int8u FlashGetSectorIdx (blt_addr address)`
Determines the index into the flashLayout[] array of the flash sector that the specified address is in.
- `static blt_addr FlashTranslateToNonCachedAddress (blt_addr address)`
The TC2 has its PFLASH accessible in the memory map in two regions. One is the non-cached region starting at FLASH_UNCACHED_BASE_ADDR and the other is the cached region starting at FLASH_CACHED_BASE_ADDR. Flash erase and programming operations need to operate on addresses in the non-cached region. It is possible that the caller of this driver's API functions, specifies memory addresses in the cached region. This function automatically translates the memory address from cached to non-cached.

- void **FlashInit** (void)
Initializes the flash driver.
- **blt_bool FlashWrite** (**blt_addr** addr, **blt_int32u** len, **blt_int8u** *data)
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- **blt_bool FlashErase** (**blt_addr** addr, **blt_int32u** len)
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- **blt_bool FlashWriteChecksum** (void)
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- **blt_bool FlashVerifyChecksum** (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- **blt_bool FlashDone** (void)
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- **blt_addr Flash GetUserProgBaseAddress** (void)
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const **tFlashSector** flashLayout []
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- static **tFlashBlockInfo** blockInfo
Local variable with information about the flash block that is currently being operated on.
- static **tFlashBlockInfo** bootBlockInfo
Local variable with information about the flash boot block.

7.75.1 Detailed Description

Bootloader flash driver source file.

7.75.2 Function Documentation

7.75.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.75.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.75.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.75.2.4 FlashEraseLogicalSector()

```
static BLT_RAM_FUNC_BEGIN blt_bool FlashEraseLogicalSector (
    blt_addr log_sector_base_addr ) [static]
```

Erases one logical sector starting at the specified base address.

Attention

This function must run from program scratch RAM and not from flash. As such, it should also not call any functions that are not in RAM. Calling inline functions is okay though.

Parameters

| | |
|-----------------------------|--|
| <i>log_sector_base_addr</i> | Base address of the logical sector to erase. |
|-----------------------------|--|

Returns

BLT_TRUE if the logical sector was successfully erased, BLT_FALSE otherwise.

Referenced by [FlashEraseSectors\(\)](#).

7.75.2.5 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector_idx,
    blt_int8u last_sector_idx ) [static]
```

Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.

Parameters

| | |
|-------------------------|---|
| <i>first_sector_idx</i> | First flash sector number index into flashLayout[]. |
| <i>last_sector_idx</i> | Last flash sector number index into flashLayout[]. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.75.2.6 FlashGetSectorIdx()

```
static BLT_RAM_FUNC_END blt_int8u FlashGetSectorIdx (
    blt_addr address ) [static]
```

Determines the index into the flashLayout[] array of the flash sector that the specified address is in.

Parameters

| | |
|----------------------|------------------------------|
| <code>address</code> | Address in the flash sector. |
|----------------------|------------------------------|

Returns

Flash sector index in `flashLayout[]` or `FLASH_INVALID_SECTOR_IDX`.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.75.2.7 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the `flashLayout` table.

Returns

Base address.

7.75.2.8 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.75.2.9 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the `block->data` and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.75.2.10 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is now being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.75.2.11 FlashTranslateToNonCachedAddress()

```
static blt_addr FlashTranslateToNonCachedAddress (
    blt_addr address ) [static]
```

The TC2 has its PFLASH accessible in the memory map in two regions. One is the non-cached region starting at FLASH_UNCACHED_BASE_ADDR and the other is the cached region starting at FLASH_CACHED_BASE_ADDR. Flash erase and programming operations need to operate on addresses in the non-cached region. It is possible that the caller of this driver's API functions, specifies memory addresses in the cached region. This function automatically translates the memory address from cached to non-cached.

Parameters

| | |
|----------------|-----------------------|
| <i>address</i> | Address to translate. |
|----------------|-----------------------|

Returns

Translated address.

Referenced by [FlashErase\(\)](#), and [FlashWrite\(\)](#).

7.75.2.12 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.75.2.13 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.75.2.14 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.75.2.15 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.75.2.16 FlashWritePage()

```
BLT_RAM_FUNC_END static BLT_RAM_FUNC_BEGIN blt_bool FlashWritePage (
    blt_addr page_base_addr,
    blt_int8u const * page_data ) [static]
```

Programs data to a flash page starting at the specified base address.

Attention

This function must run from program scratch RAM and not from flash. As such, it should also not call any functions that are not in RAM. Calling inline functions is okay though.

Parameters

| | |
|-----------------------|---|
| <i>page_base_addr</i> | Base address of the flash page. |
| <i>page_data</i> | Pointer to the byte array with data to program to the flash page. |

Returns

BLT_TRUE if the page was successfully programmed, BLT_FALSE otherwise.

Referenced by [FlashWriteBlock\(\)](#).

7.75.3 Variable Documentation

7.75.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.75.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.75.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory.

Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

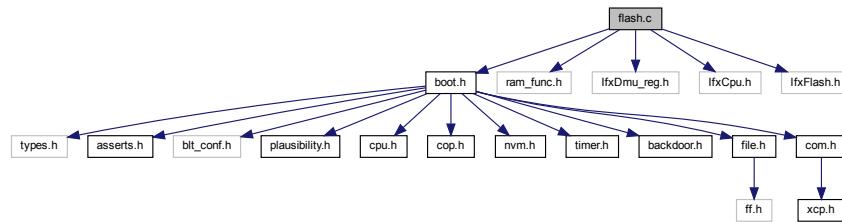
Referenced by [FlashEraseSectors\(\)](#), [FlashGetSectorIdx\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.76 flash.c File Reference

Bootloader flash driver source file.

```
#include "boot.h"
#include "ram_func.h"
#include "IfxDmu_reg.h"
#include "IfxCpu.h"
#include "IfxFlash.h"

Include dependency graph for TRICORE_TC3/flash.c:
```



Data Structures

- struct **tFlashSector**
Flash sector descriptor type.
- struct **tFlashBlockInfo**
Structure type for grouping flash block information.

Macros

- #define **FLASH_INVALID_SECTOR_IDX** (0xff)
Value for an invalid sector entry index into flashLayout[].
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_ERASE_BLOCK_SIZE** (16 * 1024)
Minimum erase size in bytes as defined by the hardware (logical sector).
- #define **FLASH_TOTAL_SECTORS** (sizeof(**flashLayout**)/sizeof(**flashLayout**[0]))
Total numbers of sectors in array flashLayout[].
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x1C)
Offset into the user program where the checksum is located. For this target it is set to the last 32-bits of the 32 byte (0x20) section at the start of the user program, which is meant for the reset handler. The reset handler doesn't need the full 32 bytes that's reserved for it. Therefore this section can be shrunk in the user program's linker script, to only be 28 bytes (0x1C) in size. This then makes 4 bytes (32-bits) available for storing the bootloader's signature checksum placeholder. Note that this macro value can be overridden in blt_conf.h, in case you want to reserve space for the signature checksum at a different memory location. Just make sure it is located in the first FLASH_WRITE_BLOCK_SIZE bytes of the user program. When changing this value, don't forget to update the location where you reserve space for the signature checksum in the user program accordingly. Otherwise the bootloader might overwrite important program code with the calculated signature checksum value, which can result in your user program not running properly.

Functions

- static `blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- static `tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- static `blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- static `blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- static `blt_bool FlashEraseSectors (blt_int8u first_sector_idx, blt_int8u last_sector_idx)`
Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.
- static `blt_int8u FlashGetSectorIdx (blt_addr address)`
Determines the index into the flashLayout[] array of the flash sector that the specified address is in.
- static `blt_bool FlashEraseLogicalSectors (blt_addr log_sector_base_addr, blt_int16u num_log_sectors)`
Erases the logical sectors starting at the specified base address.
- static `blt_bool FlashWritePage (blt_addr page_base_addr, blt_int8u const *page_data)`
Programs data to a flash page starting at the specified base address.
- void `FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Variables

- static const `tFlashSector flashLayout []`
If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.
- static `tFlashBlockInfo blockInfo`
Local variable with information about the flash block that is currently being operated on.
- static `tFlashBlockInfo bootBlockInfo`
Local variable with information about the flash boot block.

7.76.1 Detailed Description

Bootloader flash driver source file.

7.76.2 Function Documentation

7.76.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Flash destination address. |
| <i>data</i> | Pointer to the byte array with data. |
| <i>len</i> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWrite\(\)](#).

7.76.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.76.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.76.2.4 FlashEraseLogicalSectors()

```
static BLT_RAM_FUNC_BEGIN blt_bool FlashEraseLogicalSectors (
    blt_addr log_sector_base_addr,
    blt_int16u num_log_sectors ) [static]
```

Erases the logical sectors starting at the specified base address.

Attention

This function must run from program scratch RAM and not from flash. As such, it should also not call any functions that are not in RAM. Calling inline functions is okay though.

Parameters

| | |
|-----------------------------|---|
| <i>log_sector_base_addr</i> | Base address of the first logical sector. |
| <i>num_log_sectors</i> | Total number of logical sectors to erase. |

Returns

BLT_TRUE if the logical sectors were successfully erased, BLT_FALSE otherwise.

Referenced by [FlashEraseSectors\(\)](#).

7.76.2.5 FlashEraseSectors()

```
static blt_bool FlashEraseSectors (
    blt_int8u first_sector_idx,
    blt_int8u last_sector_idx ) [static]
```

Erases the flash sectors from indices first_sector_idx up until last_sector_idx into the flashLayout[] array.

Parameters

| | |
|-------------------------|---|
| <i>first_sector_idx</i> | First flash sector number index into flashLayout[]. |
| <i>last_sector_idx</i> | Last flash sector number index into flashLayout[]. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashErase\(\)](#).

7.76.2.6 FlashGetSectorIdx()

```
static BLT_RAM_FUNC_END blt_int8u FlashGetSectorIdx (
    blt_addr address ) [static]
```

Determines the index into the flashLayout[] array of the flash sector that the specified address is in.

Parameters

| | |
|----------------|------------------------------|
| <i>address</i> | Address in the flash sector. |
|----------------|------------------------------|

Returns

Flash sector index in flashLayout[] or FLASH_INVALID_SECTOR_IDX.

Referenced by [FlashErase\(\)](#), [FlashWrite\(\)](#), and [FlashWriteBlock\(\)](#).

7.76.2.7 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.76.2.8 FlashInit()

```
void FlashInit (
    void  )
```

Initializes the flash driver.

Returns

none.

7.76.2.9 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#), and [FlashSwitchBlock\(\)](#).

7.76.2.10 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is now being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.76.2.11 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.76.2.12 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.76.2.13 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#), and [FlashSwitchBlock\(\)](#).

7.76.2.14 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.76.2.15 FlashWritePage()

```
BLT_RAM_FUNC_END static BLT_RAM_FUNC_BEGIN blt_bool FlashWritePage (
    blt_addr page_base_addr,
    blt_int8u const * page_data ) [static]
```

Programs data to a flash page starting at the specified base address.

Attention

This function must run from program scratch RAM and not from flash. As such, it should also not call any functions that are not in RAM. Calling inline functions is okay though.

Parameters

| | |
|-----------------------|---|
| <i>page_base_addr</i> | Base address of the flash page. |
| <i>page_data</i> | Pointer to the byte array with data to program to the flash page. |

Returns

BLT_TRUE if the page was successfully programmed, BLT_FALSE otherwise.

Referenced by [FlashWriteBlock\(\)](#).

7.76.3 Variable Documentation

7.76.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), and [FlashWrite\(\)](#).

7.76.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), [FlashSwitchBlock\(\)](#), [FlashWrite\(\)](#), [FlashWriteBlock\(\)](#), and [FlashWriteChecksum\(\)](#).

7.76.3.3 flashLayout

```
const tFlashSector flashLayout[] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array with the layout of the flash memory.

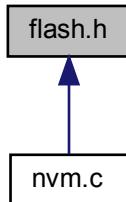
Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten.

Referenced by [FlashEraseSectors\(\)](#), [FlashGetSectorIdx\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.77 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- void [FlashInit](#) (void)
Initializes the flash driver.
- [blt_bool FlashWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- [blt_bool FlashErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- [blt_bool FlashWriteChecksum](#) (void)
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- [blt_bool FlashVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_bool FlashDone](#) (void)
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- [blt_addr Flash GetUserProgBaseAddress](#) (void)
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.77.1 Detailed Description

Bootloader flash driver header file.

7.77.2 Function Documentation

7.77.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.77.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.77.2.3 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.77.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.77.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.77.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.77.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

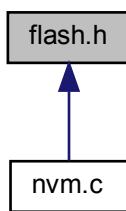
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.78 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.78.1 Detailed Description

Bootloader flash driver header file.

7.78.2 Function Documentation

7.78.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.78.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.78.2.3 FlashGetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.78.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.78.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.78.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.78.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

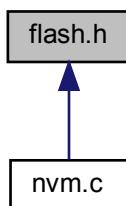
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.79 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.79.1 Detailed Description

Bootloader flash driver header file.

7.79.2 Function Documentation

7.79.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.79.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.79.2.3 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.79.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.79.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.79.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.79.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

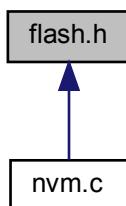
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.80 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- **void FlashInit (void)**
Initializes the flash driver.
- **blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)**
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- **blt_bool FlashErase (blt_addr addr, blt_int32u len)**
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- **blt_bool FlashWriteChecksum (void)**
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- **blt_bool FlashVerifyChecksum (void)**
Verifies the checksum, which indicates that a valid user program is present and can be started.
- **blt_bool FlashDone (void)**
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- **blt_addr Flash GetUserProgBaseAddress (void)**
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.80.1 Detailed Description

Bootloader flash driver header file.

7.80.2 Function Documentation

7.80.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.80.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.80.2.3 FlashGetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.80.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.80.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.80.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.80.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

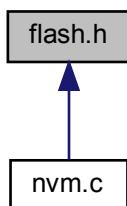
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.81 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- **void FlashInit (void)**
Initializes the flash driver.
- **blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)**
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- **blt_bool FlashErase (blt_addr addr, blt_int32u len)**
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- **blt_bool FlashWriteChecksum (void)**
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- **blt_bool FlashVerifyChecksum (void)**
Verifies the checksum, which indicates that a valid user program is present and can be started.
- **blt_bool FlashDone (void)**
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- **blt_addr Flash GetUserProgBaseAddress (void)**
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.81.1 Detailed Description

Bootloader flash driver header file.

7.81.2 Function Documentation

7.81.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.81.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.81.2.3 FlashGetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.81.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.81.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.81.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.81.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

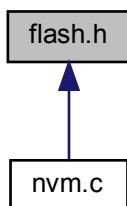
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.82 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.82.1 Detailed Description

Bootloader flash driver header file.

7.82.2 Function Documentation

7.82.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.82.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.82.2.3 FlashGetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.82.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.82.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.82.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.82.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

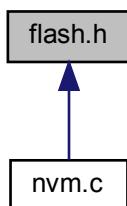
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.83 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.83.1 Detailed Description

Bootloader flash driver header file.

7.83.2 Function Documentation

7.83.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.83.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.83.2.3 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.83.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.83.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.83.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.83.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

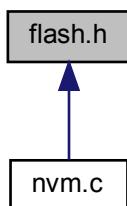
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.84 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- **void FlashInit (void)**
Initializes the flash driver.
- **blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)**
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- **blt_bool FlashErase (blt_addr addr, blt_int32u len)**
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- **blt_bool FlashWriteChecksum (void)**
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- **blt_bool FlashVerifyChecksum (void)**
Verifies the checksum, which indicates that a valid user program is present and can be started.
- **blt_bool FlashDone (void)**
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- **blt_addr Flash GetUserProgBaseAddress (void)**
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.84.1 Detailed Description

Bootloader flash driver header file.

7.84.2 Function Documentation

7.84.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.84.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.84.2.3 FlashGetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.84.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.84.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.84.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.84.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

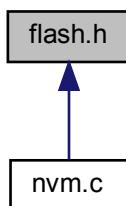
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.85 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.85.1 Detailed Description

Bootloader flash driver header file.

7.85.2 Function Documentation

7.85.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.85.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.85.2.3 FlashGetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.85.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.85.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.85.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.85.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

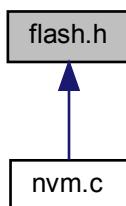
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.86 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.86.1 Detailed Description

Bootloader flash driver header file.

7.86.2 Function Documentation

7.86.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.86.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.86.2.3 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.86.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.86.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.86.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.86.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

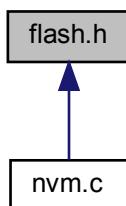
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.87 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- **void FlashInit (void)**
Initializes the flash driver.
- **blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)**
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- **blt_bool FlashErase (blt_addr addr, blt_int32u len)**
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- **blt_bool FlashWriteChecksum (void)**
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- **blt_bool FlashVerifyChecksum (void)**
Verifies the checksum, which indicates that a valid user program is present and can be started.
- **blt_bool FlashDone (void)**
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- **blt_addr Flash GetUserProgBaseAddress (void)**
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.87.1 Detailed Description

Bootloader flash driver header file.

7.87.2 Function Documentation

7.87.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.87.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.87.2.3 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.87.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.87.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.87.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.87.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

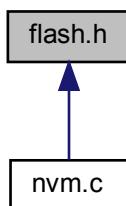
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.88 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.88.1 Detailed Description

Bootloader flash driver header file.

7.88.2 Function Documentation

7.88.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.88.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.88.2.3 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.88.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.88.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.88.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.88.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

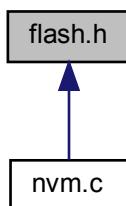
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.89 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.89.1 Detailed Description

Bootloader flash driver header file.

7.89.2 Function Documentation

7.89.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.89.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.89.2.3 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.89.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.89.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.89.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.89.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

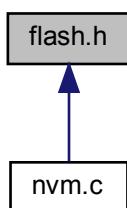
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.90 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.90.1 Detailed Description

Bootloader flash driver header file.

7.90.2 Function Documentation

7.90.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.90.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.90.2.3 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.90.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.90.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.90.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.90.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

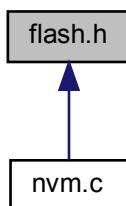
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.91 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- **void FlashInit (void)**
Initializes the flash driver.
- **blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)**
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- **blt_bool FlashErase (blt_addr addr, blt_int32u len)**
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- **blt_bool FlashWriteChecksum (void)**
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- **blt_bool FlashVerifyChecksum (void)**
Verifies the checksum, which indicates that a valid user program is present and can be started.
- **blt_bool FlashDone (void)**
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- **blt_addr Flash GetUserProgBaseAddress (void)**
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.91.1 Detailed Description

Bootloader flash driver header file.

7.91.2 Function Documentation

7.91.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.91.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.91.2.3 FlashGetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.91.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.91.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.91.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.91.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

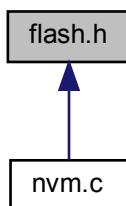
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.92 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.92.1 Detailed Description

Bootloader flash driver header file.

7.92.2 Function Documentation

7.92.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.92.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.92.2.3 FlashGetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.92.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.92.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.92.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.92.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

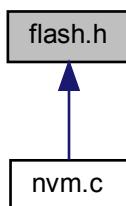
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.93 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.93.1 Detailed Description

Bootloader flash driver header file.

7.93.2 Function Documentation

7.93.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.93.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.93.2.3 FlashGetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.93.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.93.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.93.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.93.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

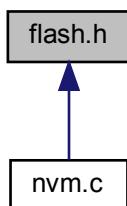
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.94 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.94.1 Detailed Description

Bootloader flash driver header file.

7.94.2 Function Documentation

7.94.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.94.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.94.2.3 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.94.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.94.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.94.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.94.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

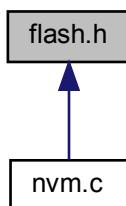
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.95 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.95.1 Detailed Description

Bootloader flash driver header file.

7.95.2 Function Documentation

7.95.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.95.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.95.2.3 FlashGetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.95.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.95.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.95.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.95.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

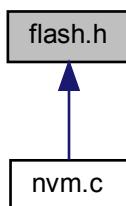
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.96 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.96.1 Detailed Description

Bootloader flash driver header file.

7.96.2 Function Documentation

7.96.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.96.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.96.2.3 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.96.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.96.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.96.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.96.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

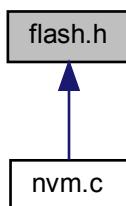
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.97 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.97.1 Detailed Description

Bootloader flash driver header file.

7.97.2 Function Documentation

7.97.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.97.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.97.2.3 FlashGetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.97.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.97.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.97.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.97.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

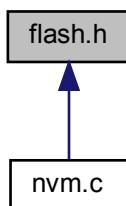
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.98 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.98.1 Detailed Description

Bootloader flash driver header file.

7.98.2 Function Documentation

7.98.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.98.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.98.2.3 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.98.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.98.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.98.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.98.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

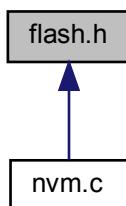
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.99 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.99.1 Detailed Description

Bootloader flash driver header file.

7.99.2 Function Documentation

7.99.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.99.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.99.2.3 FlashGetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.99.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.99.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.99.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.99.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

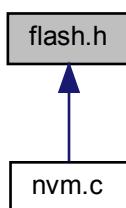
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.100 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.100.1 Detailed Description

Bootloader flash driver header file.

7.100.2 Function Documentation

7.100.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.100.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.100.2.3 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.100.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.100.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.100.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.100.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

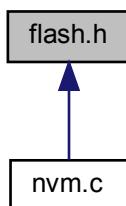
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.101 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.101.1 Detailed Description

Bootloader flash driver header file.

7.101.2 Function Documentation

7.101.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.101.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.101.2.3 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.101.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.101.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.101.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.101.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

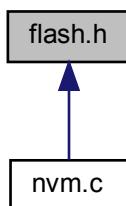
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.102 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.102.1 Detailed Description

Bootloader flash driver header file.

7.102.2 Function Documentation

7.102.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.102.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.102.2.3 FlashGetUserProgBaseAddress()

```
blt_addr FlashGetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.102.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.102.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.102.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.102.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

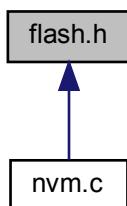
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.103 flash.h File Reference

Bootloader flash driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

7.103.1 Detailed Description

Bootloader flash driver header file.

7.103.2 Function Documentation

7.103.2.1 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.103.2.2 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.103.2.3 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

Referenced by [Nvm GetUserProgBaseAddress\(\)](#).

7.103.2.4 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

Attention

This entire flash driver should not use the FLASH_SIZE or FLASH_BANK_SIZE macros. These macros indirectly access FLASHSIZE_BASE (0x08FFF80CUL), which might be in a memory region that requires an MPU configuration to be able to access it. Use macro BOOT_NVM_SIZE_KB instead to access the same information, taking into account that there are always two flash banks.

On top of that, there is an errata (2.2.22) where some early STM32H562/3 devices have the wrong flash size configured, meaning that the FLASH_SIZE and FLASH_BANK_SIZE macros cannot be relied on in this STM32H5 port.

Note that this flash driver assumes that the instruction cache (ICache) is enabled. Otherwise an NMI might trigger during flash erase and programming operations. Note that the CPU module manages the enabled/disabling of the instruction cache.

Returns

none.

7.103.2.5 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [NvmVerifyChecksum\(\)](#).

7.103.2.6 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashWriteChecksum\(\)](#).

7.103.2.7 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

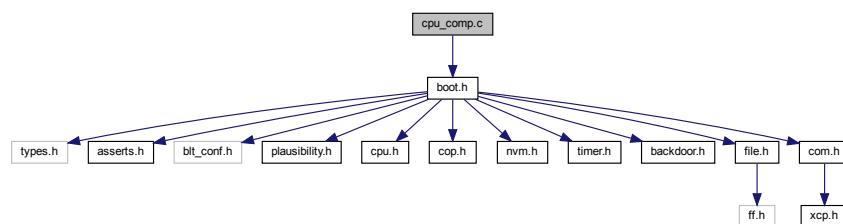
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.104 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for _template/GCC/cpu_comp.c:
```

**Functions**

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.104.1 Detailed Description

Bootloader cpu module source file.

7.104.2 Function Documentation

7.104.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

Referenced by [CpuInit\(\)](#).

7.104.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

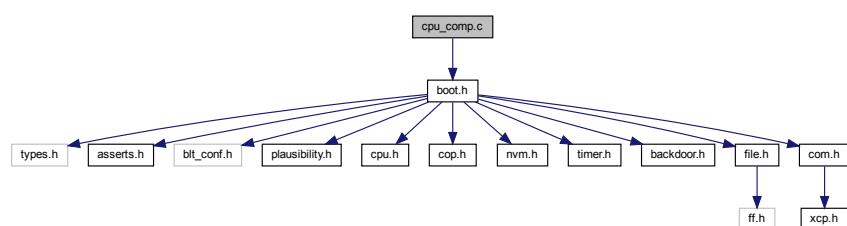
Returns

none.

7.105 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_S32K11/GCC/cpu_comp.c:
```



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.105.1 Detailed Description

Bootloader cpu module source file.

7.105.2 Function Documentation

7.105.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.105.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

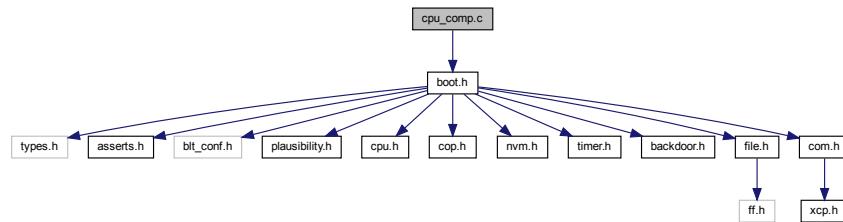
Returns

none.

7.106 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_S32K11/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.106.1 Detailed Description

Bootloader cpu module source file.

7.106.2 Function Documentation

7.106.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.106.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

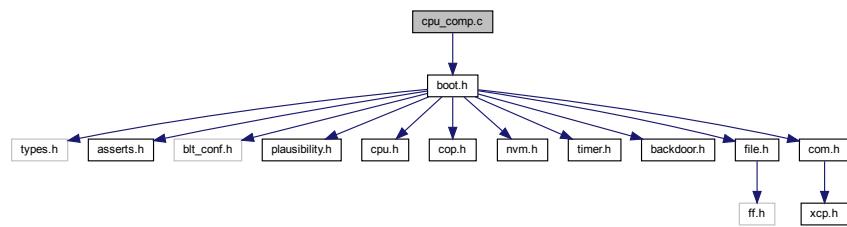
none.

7.107 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM0_STM32C0/GCC/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.107.1 Detailed Description

Bootloader cpu module source file.

7.107.2 Function Documentation

7.107.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.107.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

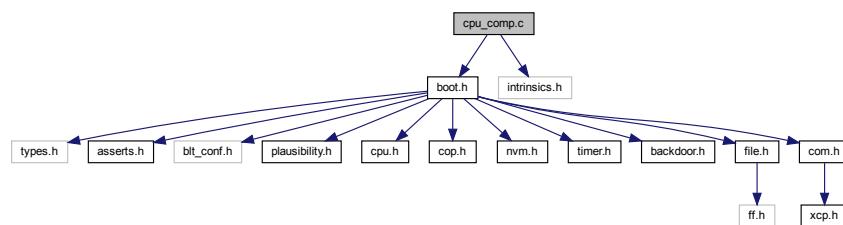
Returns

none.

7.108 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM0_STM32C0/IAR/cpu_comp.c:
```



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.108.1 Detailed Description

Bootloader cpu module source file.

7.108.2 Function Documentation

7.108.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.108.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

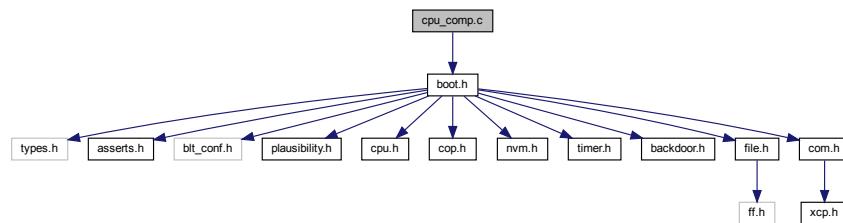
Returns

none.

7.109 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_STM32C0/Keil/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.109.1 Detailed Description

Bootloader cpu module source file.

7.109.2 Function Documentation

7.109.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.109.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

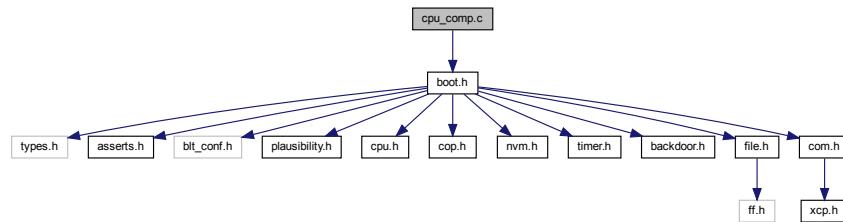
Returns

none.

7.110 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_STM32F0/GCC/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.110.1 Detailed Description

Bootloader cpu module source file.

7.110.2 Function Documentation

7.110.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.110.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

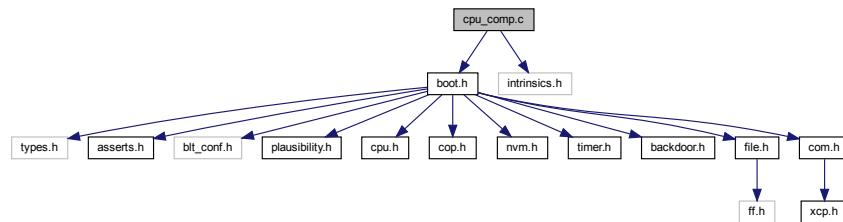
Returns

none.

7.111 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM0_STM32F0/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.111.1 Detailed Description

Bootloader cpu module source file.

7.111.2 Function Documentation

7.111.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.111.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

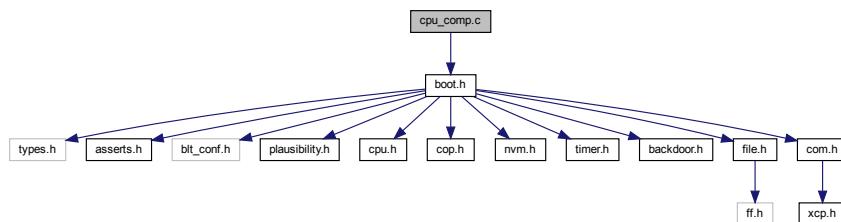
Returns

none.

7.112 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_STM32F0/Keil/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.112.1 Detailed Description

Bootloader cpu module source file.

7.112.2 Function Documentation

7.112.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.112.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

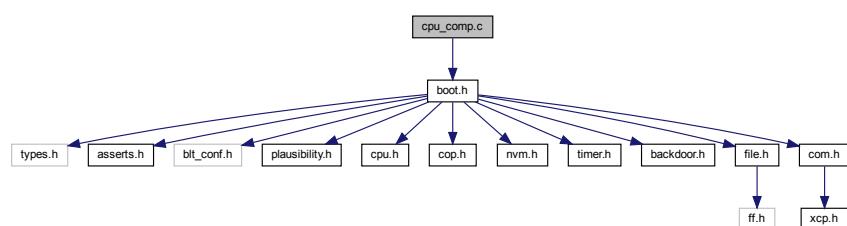
none.

7.113 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM0_STM32G0/GCC/cpu_comp.c:



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.113.1 Detailed Description

Bootloader cpu module source file.

7.113.2 Function Documentation

7.113.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.113.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

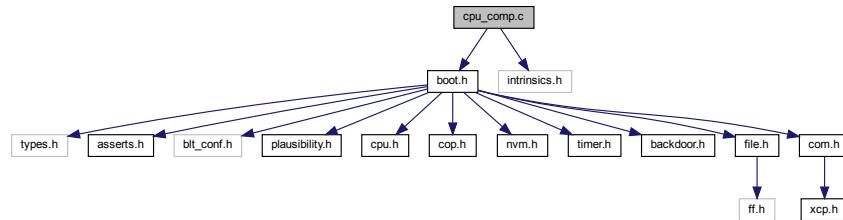
Returns

none.

7.114 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM0_STM32G0/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.114.1 Detailed Description

Bootloader cpu module source file.

7.114.2 Function Documentation

7.114.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.114.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

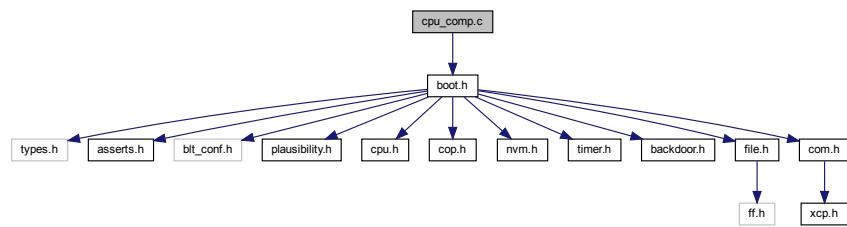
Returns

none.

7.115 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_STM32G0/Keil/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.115.1 Detailed Description

Bootloader cpu module source file.

7.115.2 Function Documentation

7.115.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.115.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

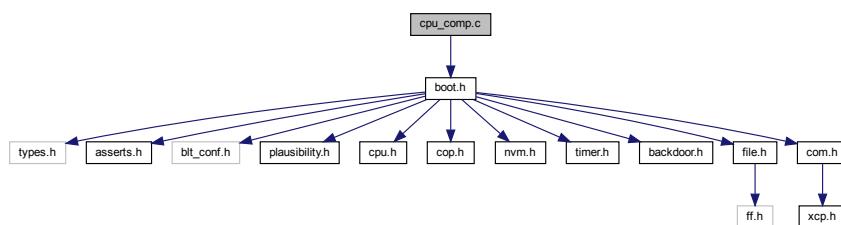
none.

7.116 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM0_STM32L0/GCC/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.116.1 Detailed Description

Bootloader cpu module source file.

7.116.2 Function Documentation

7.116.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.116.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

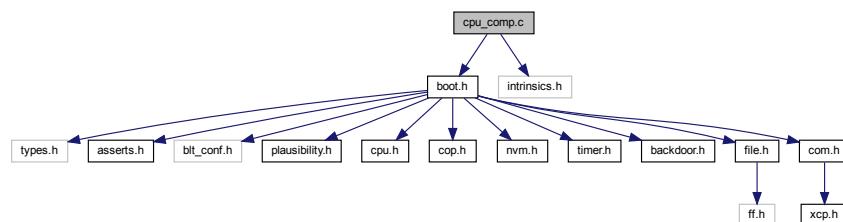
Returns

none.

7.117 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM0_STM32L0/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.117.1 Detailed Description

Bootloader cpu module source file.

7.117.2 Function Documentation

7.117.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.117.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

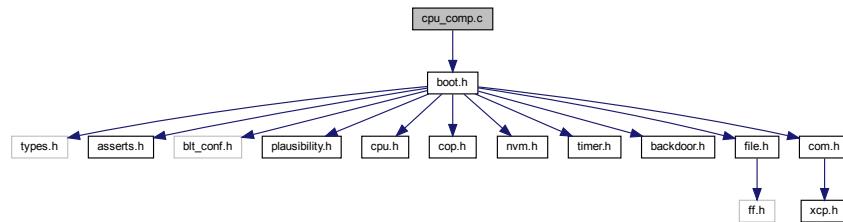
Returns

none.

7.118 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_STM32L0/Keil/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.118.1 Detailed Description

Bootloader cpu module source file.

7.118.2 Function Documentation

7.118.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.118.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

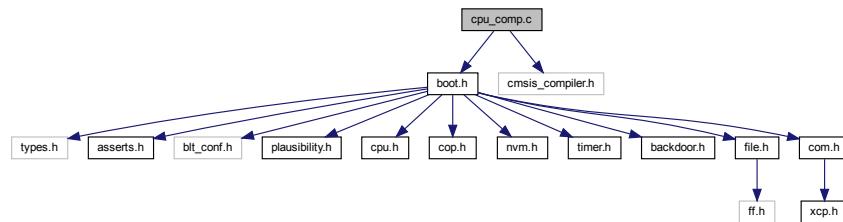
Returns

none.

7.119 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "cmsis_compiler.h"
Include dependency graph for ARMCM0_XMC1/GCC/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)

Disable global interrupts.

- void [CpuIrqEnable](#) (void)

Enable global interrupts.

7.119.1 Detailed Description

Bootloader cpu module source file.

7.119.2 Function Documentation

7.119.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.119.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

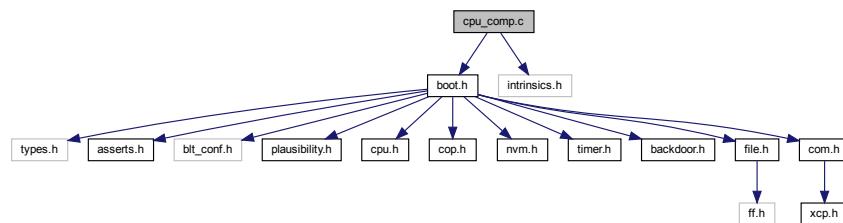
Returns

none.

7.120 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM0_XMC1/IAR/cpu_comp.c:
```



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.120.1 Detailed Description

Bootloader cpu module source file.

7.120.2 Function Documentation

7.120.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.120.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

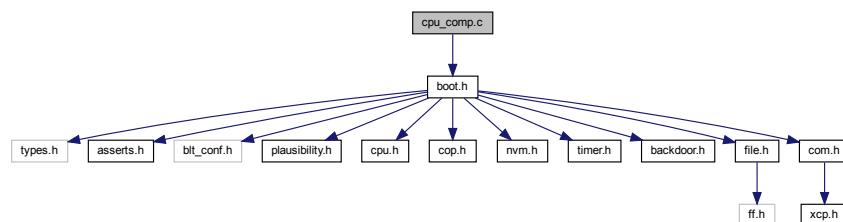
Returns

none.

7.121 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_XMC1/Keil/cpu_comp.c:
```



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.121.1 Detailed Description

Bootloader cpu module source file.

7.121.2 Function Documentation

7.121.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.121.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

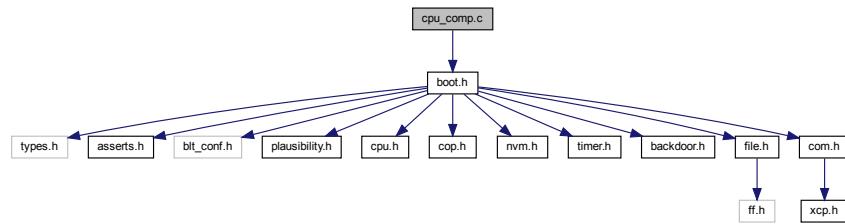
none.

7.122 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM33_STM32H5/GCC/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.122.1 Detailed Description

Bootloader cpu module source file.

7.122.2 Function Documentation

7.122.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.122.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

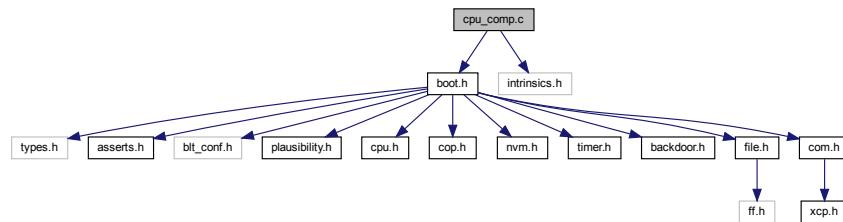
Returns

none.

7.123 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM33_STM32H5/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.123.1 Detailed Description

Bootloader cpu module source file.

7.123.2 Function Documentation

7.123.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.123.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

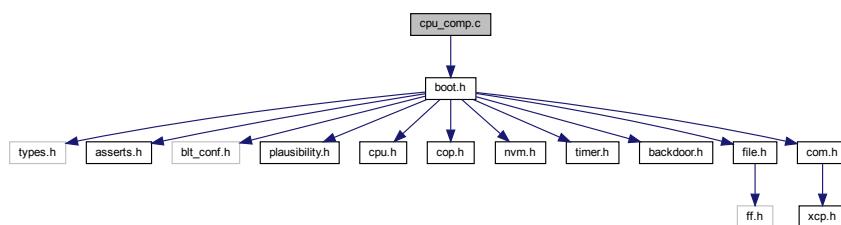
none.

7.124 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM33_STM32H5/Keil/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.124.1 Detailed Description

Bootloader cpu module source file.

7.124.2 Function Documentation

7.124.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.124.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

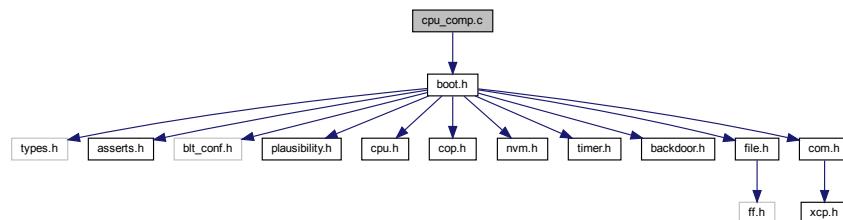
none.

7.125 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM33_STM32L5/GCC/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.125.1 Detailed Description

Bootloader cpu module source file.

7.125.2 Function Documentation

7.125.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.125.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

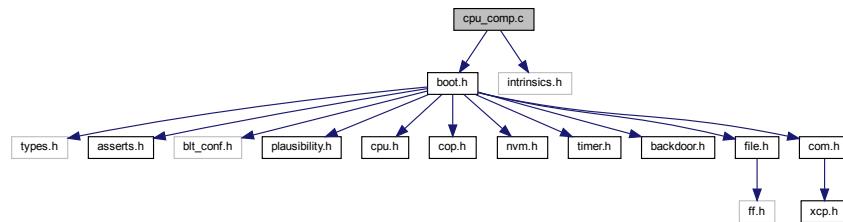
Returns

none.

7.126 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM33_STM32L5/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.126.1 Detailed Description

Bootloader cpu module source file.

7.126.2 Function Documentation

7.126.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.126.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

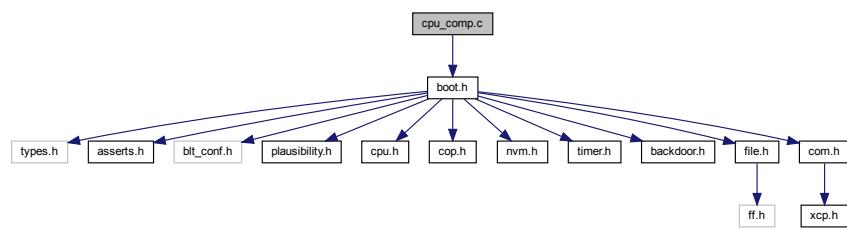
none.

7.127 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM33_STM32L5/Keil/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.127.1 Detailed Description

Bootloader cpu module source file.

7.127.2 Function Documentation

7.127.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.127.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

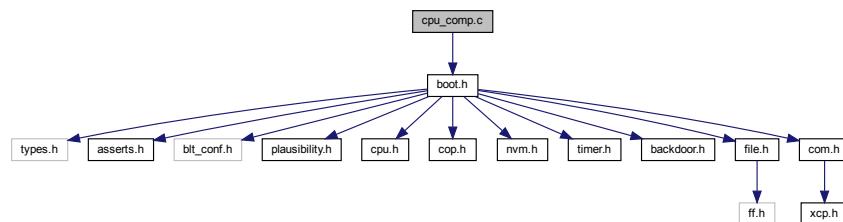
none.

7.128 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM33_STM32U5/GCC/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)

Disable global interrupts.

- void [CpuIrqEnable](#) (void)

Enable global interrupts.

7.128.1 Detailed Description

Bootloader cpu module source file.

7.128.2 Function Documentation

7.128.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.128.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

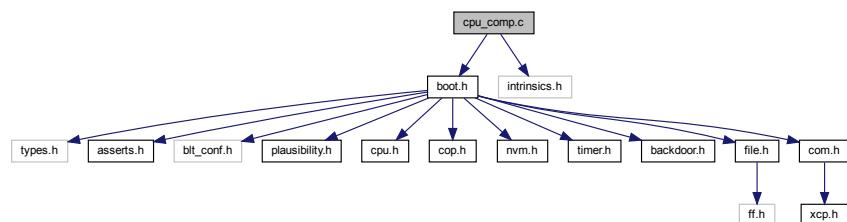
Returns

none.

7.129 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM33_STM32U5/IAR/cpu_comp.c:
```



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.129.1 Detailed Description

Bootloader cpu module source file.

7.129.2 Function Documentation

7.129.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.129.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

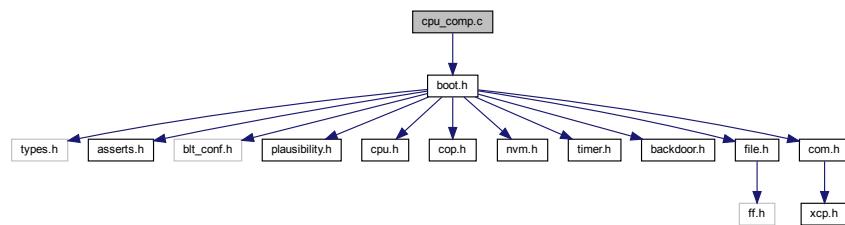
none.

7.130 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM33_STM32U5/Keil/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.130.1 Detailed Description

Bootloader cpu module source file.

7.130.2 Function Documentation

7.130.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.130.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

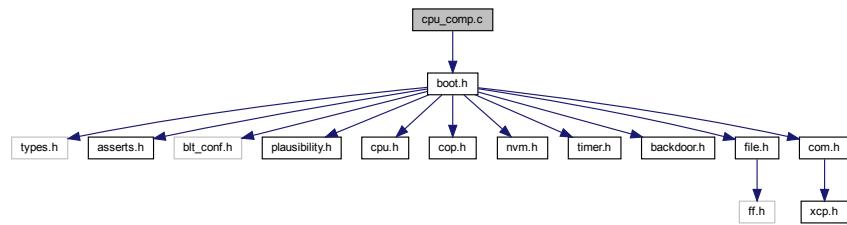
Returns

none.

7.131 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_EFM32/GCC/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.131.1 Detailed Description

Bootloader cpu module source file.

7.131.2 Function Documentation

7.131.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.131.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

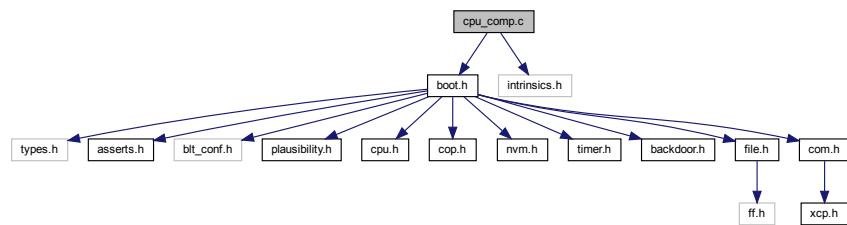
Returns

none.

7.132 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM3_EFM32/IAR/cpu_comp.c:
```



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.132.1 Detailed Description

Bootloader cpu module source file.

7.132.2 Function Documentation

7.132.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.132.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

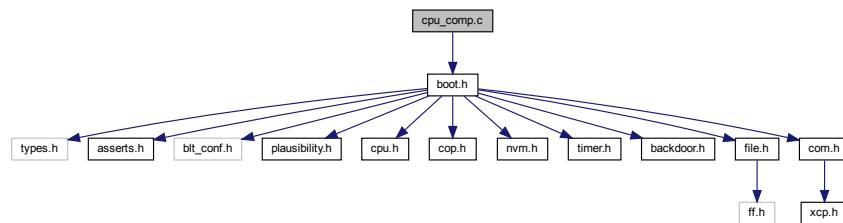
Returns

none.

7.133 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_LM3S/GCC/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.133.1 Detailed Description

Bootloader cpu module source file.

7.133.2 Function Documentation

7.133.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.133.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

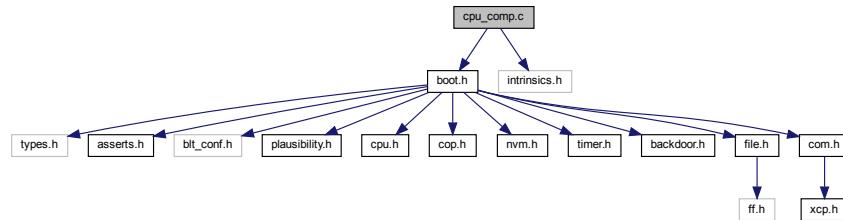
Returns

none.

7.134 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM3_LM3S/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.134.1 Detailed Description

Bootloader cpu module source file.

7.134.2 Function Documentation

7.134.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.134.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

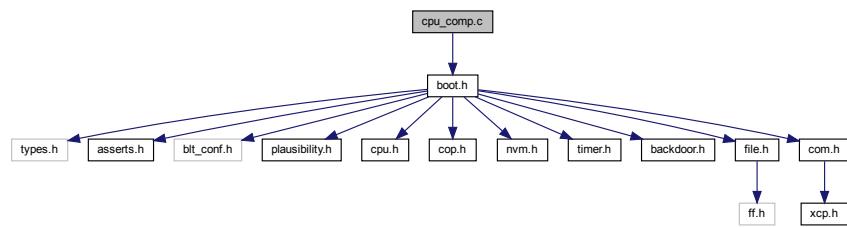
none.

7.135 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM3_STM32F1/GCC/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.135.1 Detailed Description

Bootloader cpu module source file.

7.135.2 Function Documentation

7.135.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.135.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

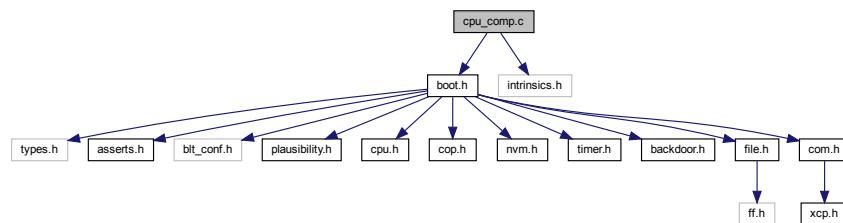
Returns

none.

7.136 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM3_STM32F1/IAR/cpu_comp.c:
```



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.136.1 Detailed Description

Bootloader cpu module source file.

7.136.2 Function Documentation

7.136.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.136.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

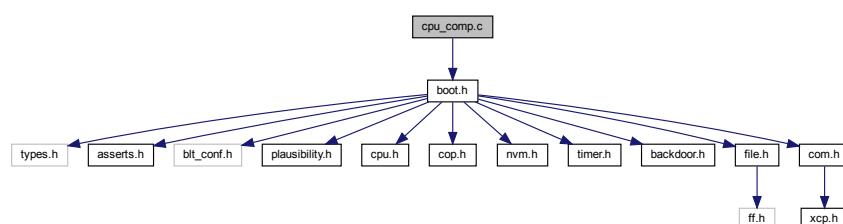
Returns

none.

7.137 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_STM32F1/Keil/cpu_comp.c:
```



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.137.1 Detailed Description

Bootloader cpu module source file.

7.137.2 Function Documentation

7.137.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.137.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

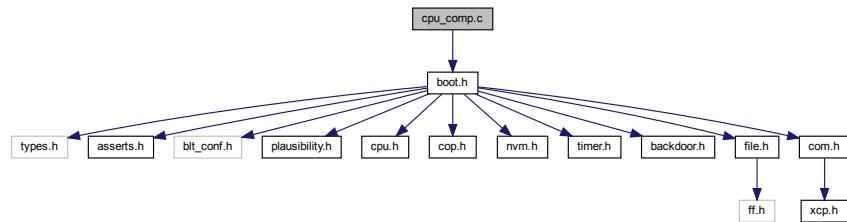
Returns

none.

7.138 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_STM32F2/GCC/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.138.1 Detailed Description

Bootloader cpu module source file.

7.138.2 Function Documentation

7.138.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.138.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

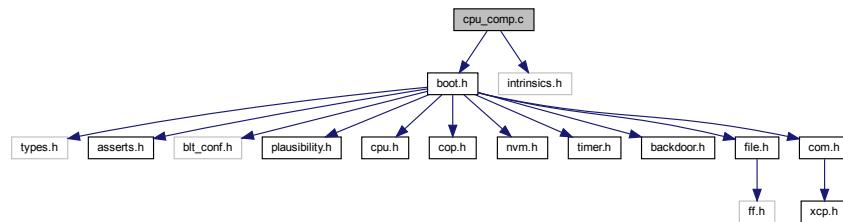
Returns

none.

7.139 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM3_STM32F2/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)

Disable global interrupts.

- void [CpuIrqEnable](#) (void)

Enable global interrupts.

7.139.1 Detailed Description

Bootloader cpu module source file.

7.139.2 Function Documentation

7.139.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.139.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

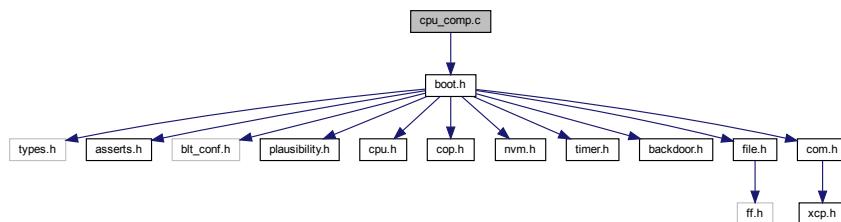
Returns

none.

7.140 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_STM32F2/Keil/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.140.1 Detailed Description

Bootloader cpu module source file.

7.140.2 Function Documentation

7.140.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.140.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

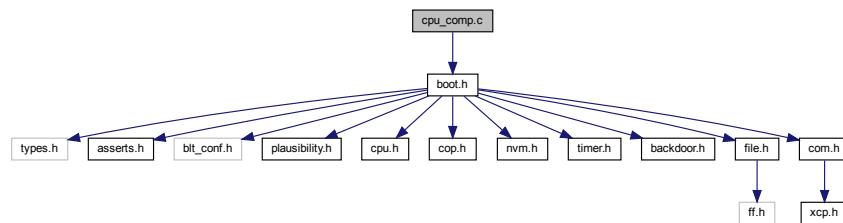
none.

7.141 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM3_STM32L1/GCC/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.141.1 Detailed Description

Bootloader cpu module source file.

7.141.2 Function Documentation

7.141.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.141.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

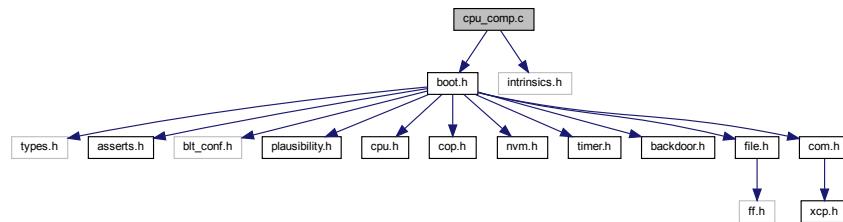
Returns

none.

7.142 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM3_STM32L1/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.142.1 Detailed Description

Bootloader cpu module source file.

7.142.2 Function Documentation

7.142.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.142.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

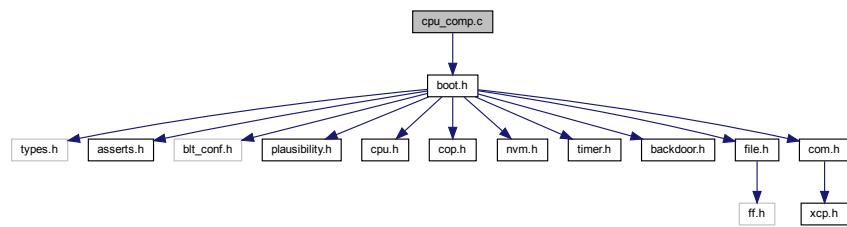
Returns

none.

7.143 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_STM32L1/Keil/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.143.1 Detailed Description

Bootloader cpu module source file.

7.143.2 Function Documentation

7.143.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.143.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

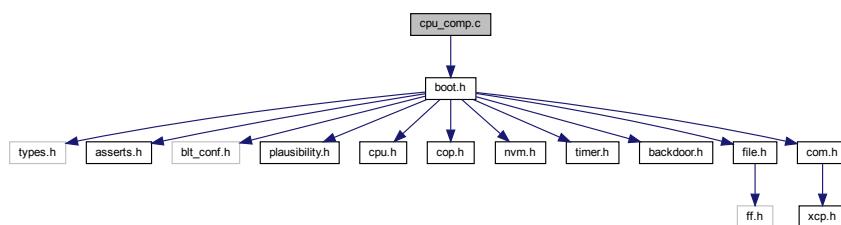
Returns

none.

7.144 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_S32K14/GCC/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.144.1 Detailed Description

Bootloader cpu module source file.

7.144.2 Function Documentation

7.144.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.144.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

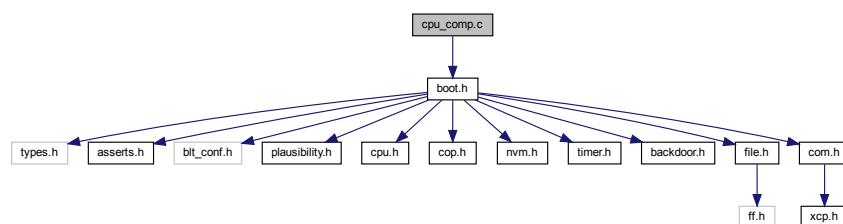
none.

7.145 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM4_S32K14/IAR/cpu_comp.c:



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.145.1 Detailed Description

Bootloader cpu module source file.

7.145.2 Function Documentation

7.145.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.145.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

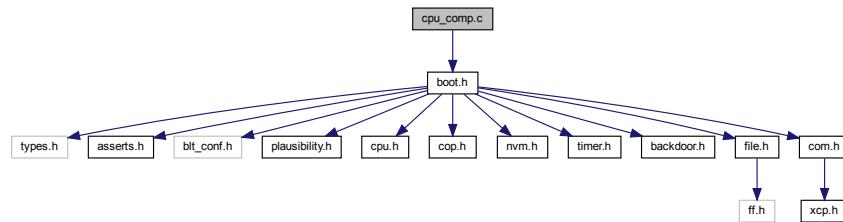
Returns

none.

7.146 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_STM32F3/GCC/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.146.1 Detailed Description

Bootloader cpu module source file.

7.146.2 Function Documentation

7.146.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.146.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

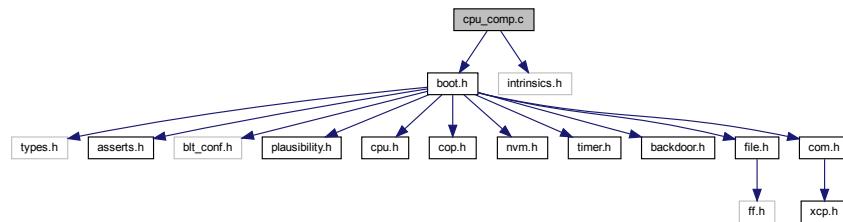
Returns

none.

7.147 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM4_STM32F3/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)

Disable global interrupts.

- void [CpuIrqEnable](#) (void)

Enable global interrupts.

7.147.1 Detailed Description

Bootloader cpu module source file.

7.147.2 Function Documentation

7.147.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.147.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

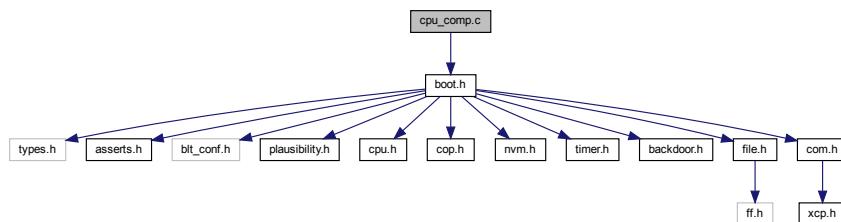
Returns

none.

7.148 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_STM32F3/Keil/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.148.1 Detailed Description

Bootloader cpu module source file.

7.148.2 Function Documentation

7.148.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.148.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

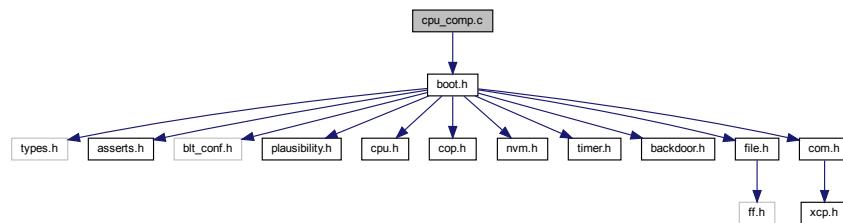
none.

7.149 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM4_STM32F4/GCC/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.149.1 Detailed Description

Bootloader cpu module source file.

7.149.2 Function Documentation

7.149.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.149.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

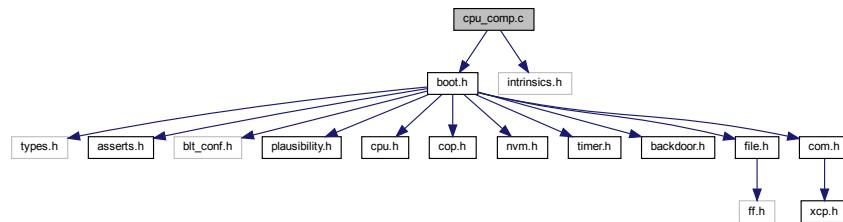
Returns

none.

7.150 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM4_STM32F4/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.150.1 Detailed Description

Bootloader cpu module source file.

7.150.2 Function Documentation

7.150.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.150.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

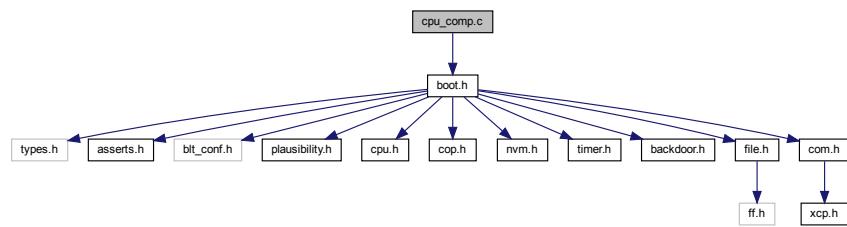
Returns

none.

7.151 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_STM32F4/Keil/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.151.1 Detailed Description

Bootloader cpu module source file.

7.151.2 Function Documentation

7.151.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.151.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

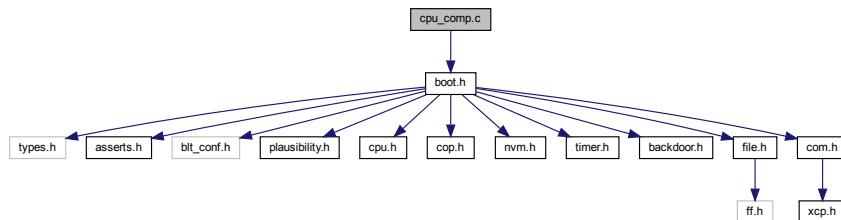
none.

7.152 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM4_STM32G4/GCC/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)

Disable global interrupts.

- void [CpuIrqEnable](#) (void)

Enable global interrupts.

7.152.1 Detailed Description

Bootloader cpu module source file.

7.152.2 Function Documentation

7.152.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.152.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

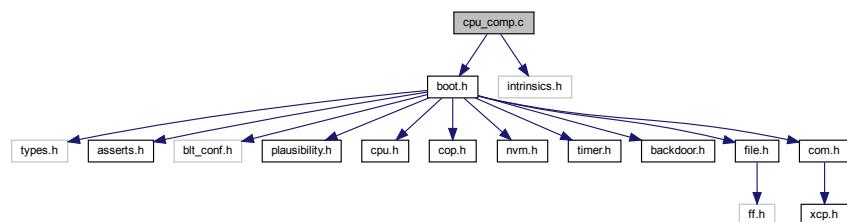
Returns

none.

7.153 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM4_STM32G4/IAR/cpu_comp.c:
```



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.153.1 Detailed Description

Bootloader cpu module source file.

7.153.2 Function Documentation

7.153.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.153.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

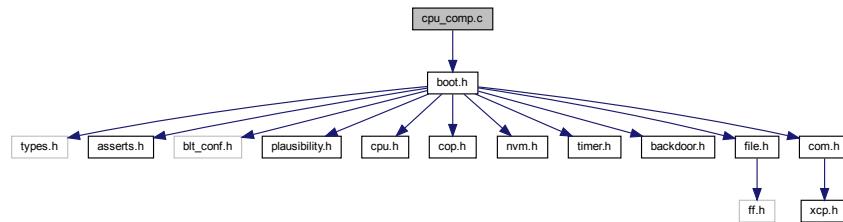
Returns

none.

7.154 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_STM32G4/Keil/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.154.1 Detailed Description

Bootloader cpu module source file.

7.154.2 Function Documentation

7.154.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.154.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

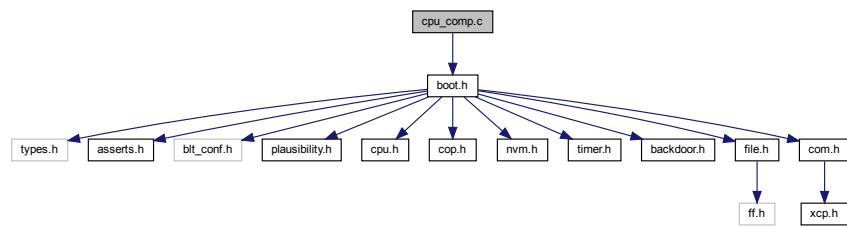
none.

7.155 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM4_STM32L4/GCC/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.155.1 Detailed Description

Bootloader cpu module source file.

7.155.2 Function Documentation

7.155.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.155.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

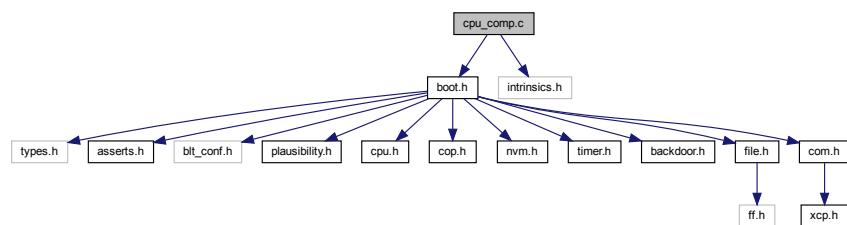
Returns

none.

7.156 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM4_STM32L4/IAR/cpu_comp.c:
```



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.156.1 Detailed Description

Bootloader cpu module source file.

7.156.2 Function Documentation

7.156.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.156.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

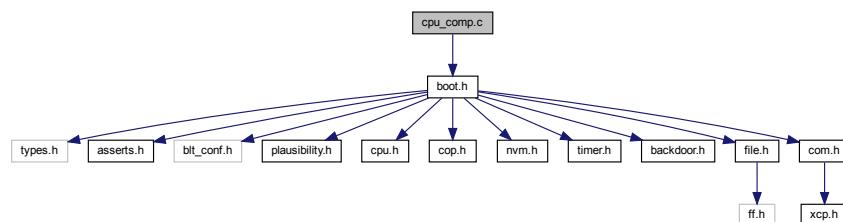
Returns

none.

7.157 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_STM32L4/Keil/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.157.1 Detailed Description

Bootloader cpu module source file.

7.157.2 Function Documentation

7.157.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.157.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

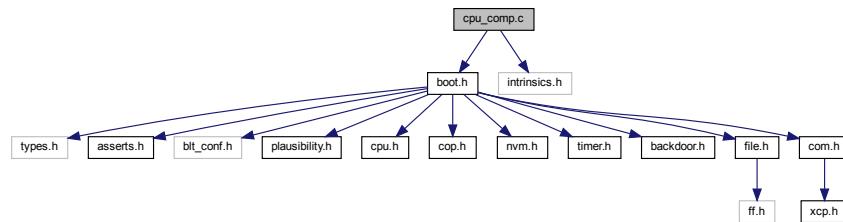
Returns

none.

7.158 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM4_TM4C/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.158.1 Detailed Description

Bootloader cpu module source file.

7.158.2 Function Documentation

7.158.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.158.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

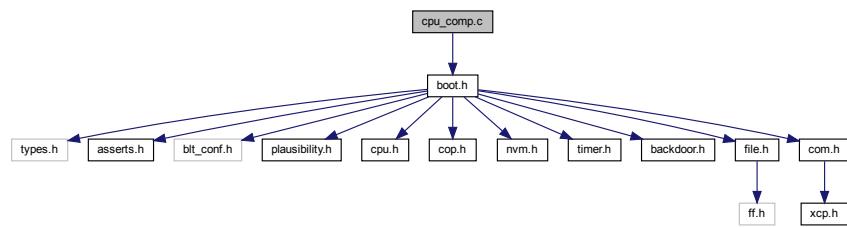
Returns

none.

7.159 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_XMC4/GCC/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.159.1 Detailed Description

Bootloader cpu module source file.

7.159.2 Function Documentation

7.159.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.159.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

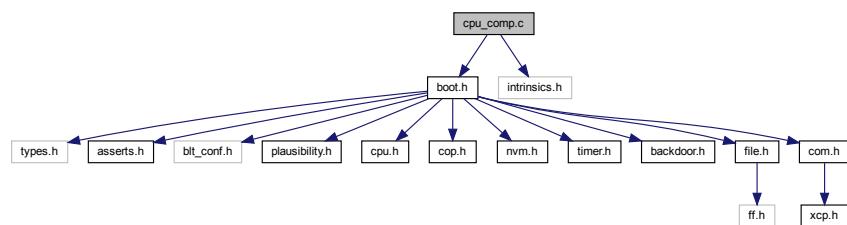
Returns

none.

7.160 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM4_XMC4/IAR/cpu_comp.c:
```



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.160.1 Detailed Description

Bootloader cpu module source file.

7.160.2 Function Documentation

7.160.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.160.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

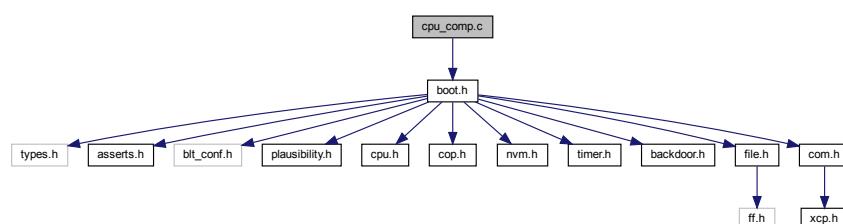
Returns

none.

7.161 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_XMC4/Keil/cpu_comp.c:
```



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.161.1 Detailed Description

Bootloader cpu module source file.

7.161.2 Function Documentation

7.161.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.161.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

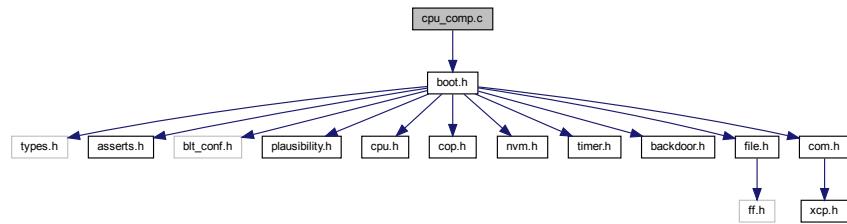
Returns

none.

7.162 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM7_STM32F7/GCC/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.162.1 Detailed Description

Bootloader cpu module source file.

7.162.2 Function Documentation

7.162.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.162.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

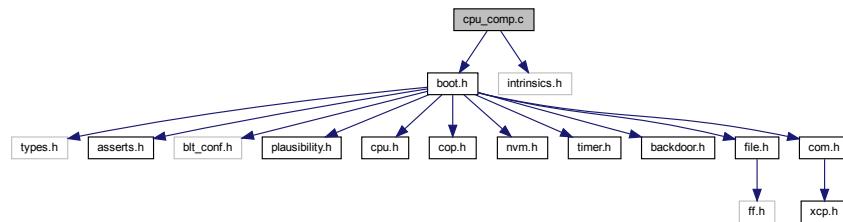
Returns

none.

7.163 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM7_STM32F7/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.163.1 Detailed Description

Bootloader cpu module source file.

7.163.2 Function Documentation

7.163.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.163.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

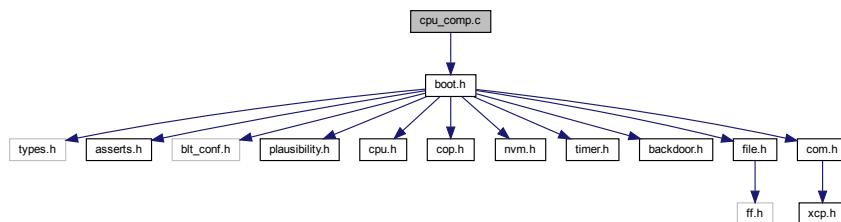
Returns

none.

7.164 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM7_STM32F7/Keil/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.164.1 Detailed Description

Bootloader cpu module source file.

7.164.2 Function Documentation

7.164.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.164.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

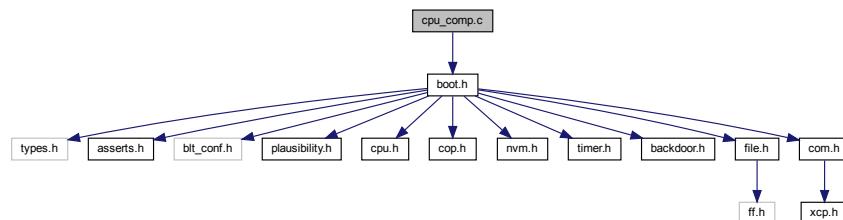
none.

7.165 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM7_STM32H7/GCC/cpu_comp.c:



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.165.1 Detailed Description

Bootloader cpu module source file.

7.165.2 Function Documentation

7.165.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.165.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

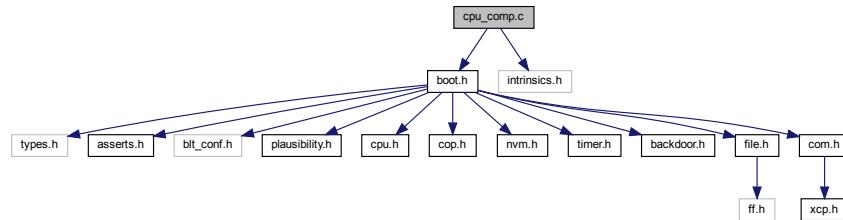
Returns

none.

7.166 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include <intrinsics.h>
Include dependency graph for ARMCM7_STM32H7/IAR/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.166.1 Detailed Description

Bootloader cpu module source file.

7.166.2 Function Documentation

7.166.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.166.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

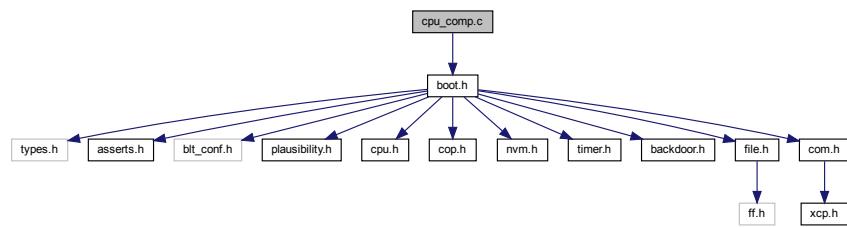
Returns

none.

7.167 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for ARMCM7_STM32H7/Keil/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.167.1 Detailed Description

Bootloader cpu module source file.

7.167.2 Function Documentation

7.167.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.167.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

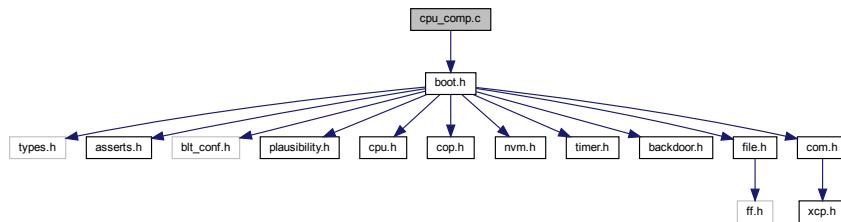
Returns

none.

7.168 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
Include dependency graph for HCS12/CodeWarrior/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.168.1 Detailed Description

Bootloader cpu module source file.

7.168.2 Function Documentation

7.168.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.168.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

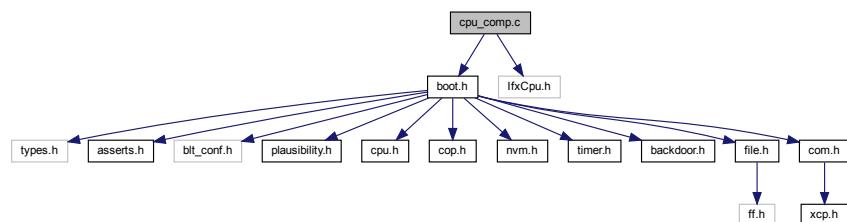
Returns

none.

7.169 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "IfxCpu.h"
Include dependency graph for TRICORE_TC2/Tasking/cpu_comp.c:
```



Functions

- void **CpuIrqDisable** (void)
Disable global interrupts.
- void **CpuIrqEnable** (void)
Enable global interrupts.

7.169.1 Detailed Description

Bootloader cpu module source file.

7.169.2 Function Documentation

7.169.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.169.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

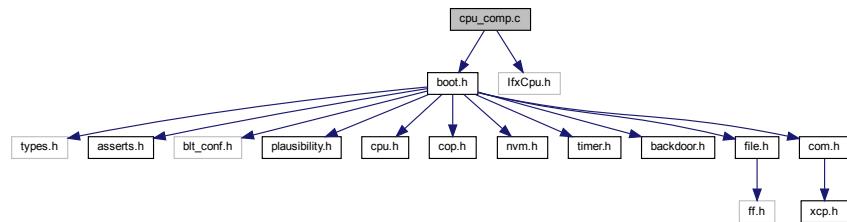
Returns

none.

7.170 cpu_comp.c File Reference

Bootloader cpu module source file.

```
#include "boot.h"
#include "IfxCpu.h"
Include dependency graph for TRICORE_TC3/Tasking/cpu_comp.c:
```



Functions

- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.170.1 Detailed Description

Bootloader cpu module source file.

7.170.2 Function Documentation

7.170.2.1 CpuIrqDisable()

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

7.170.2.2 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

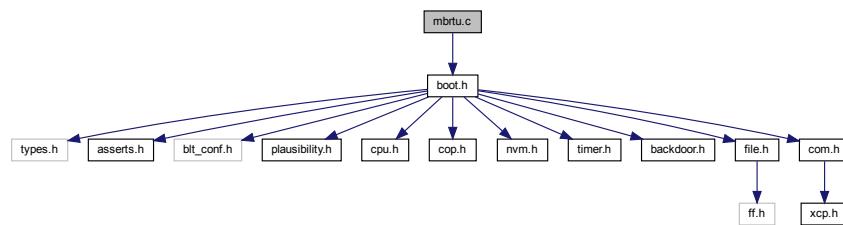
Returns

none.

7.171 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for _template/mbrtu.c:
```



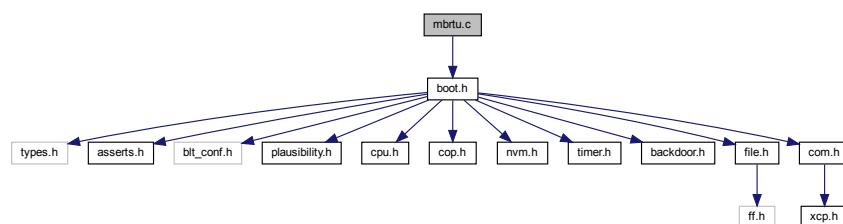
7.171.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.172 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_S32K11/mbrtu.c:
```



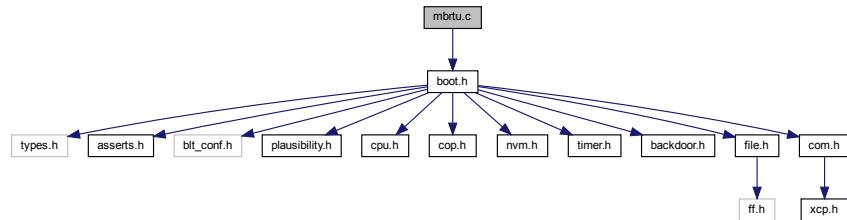
7.172.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.173 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_STM32C0/mbrtu.c:
```



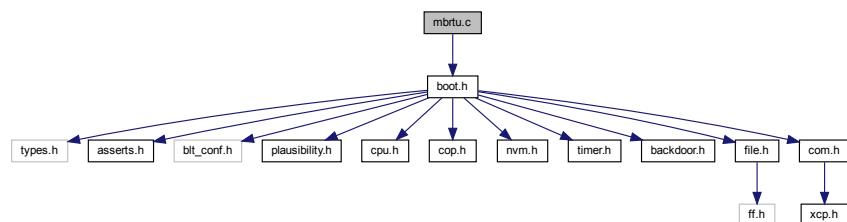
7.173.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.174 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_STM32F0/mbrtu.c:
```



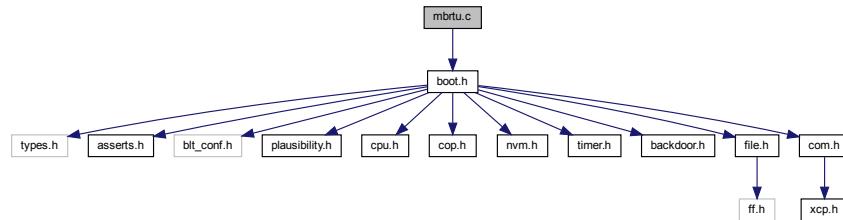
7.174.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.175 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_STM32G0/mbrtu.c:
```



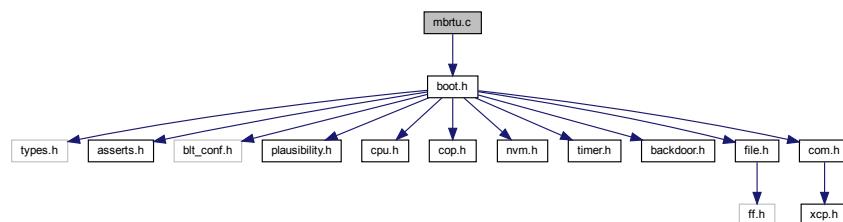
7.175.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.176 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_STM32L0/mbrtu.c:
```



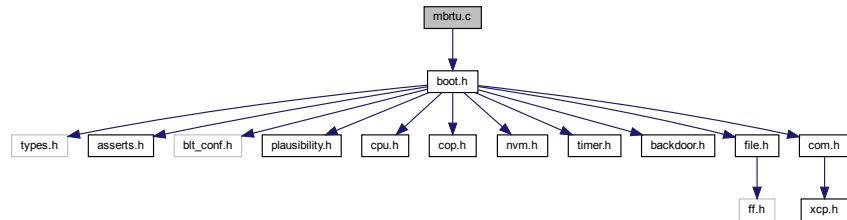
7.176.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.177 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_XMC1/mbrtu.c:
```



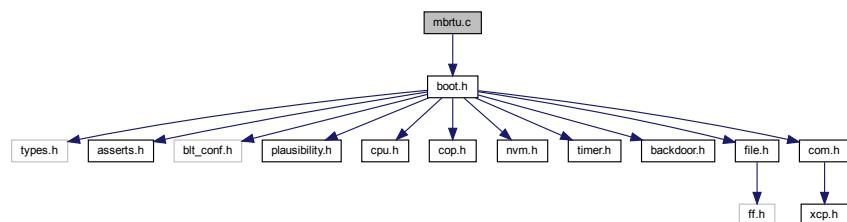
7.177.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.178 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM33_STM32H5/mbrtu.c:
```



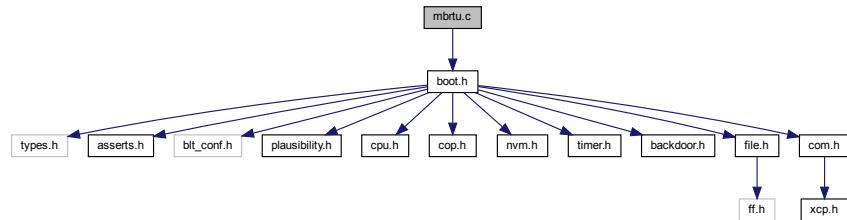
7.178.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.179 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM33_STM32L5/mbrtu.c:
```



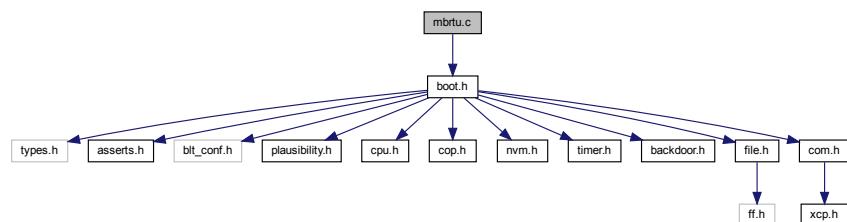
7.179.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.180 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM33_STM32U5/mbrtu.c:
```



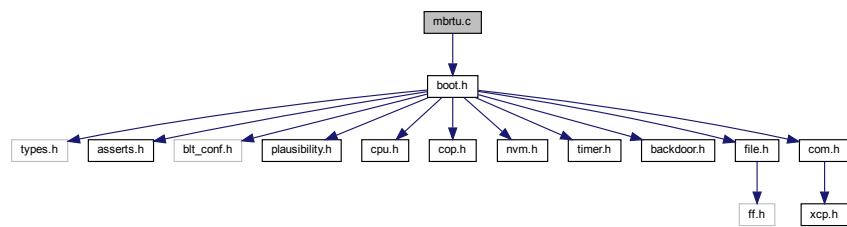
7.180.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.181 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_EFM32/mbrtu.c:
```



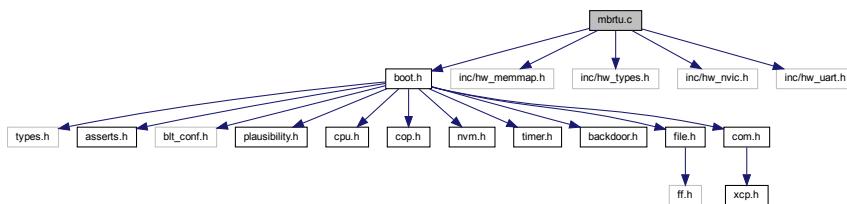
7.181.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.182 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_nvic.h"
#include "inc/hw_uart.h"
Include dependency graph for ARMCM3_LM3S/mbrtu.c:
```



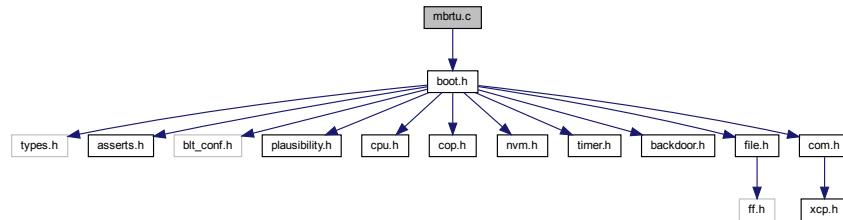
7.182.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.183 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_STM32F1/mbrtu.c:
```



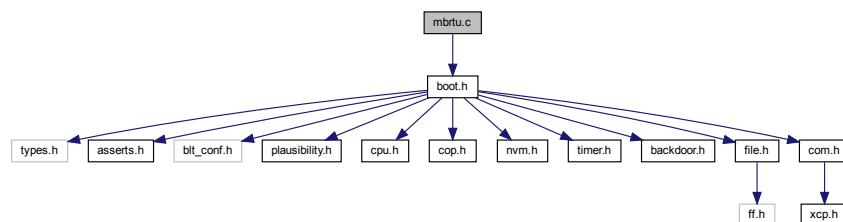
7.183.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.184 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_STM32F2/mbrtu.c:
```



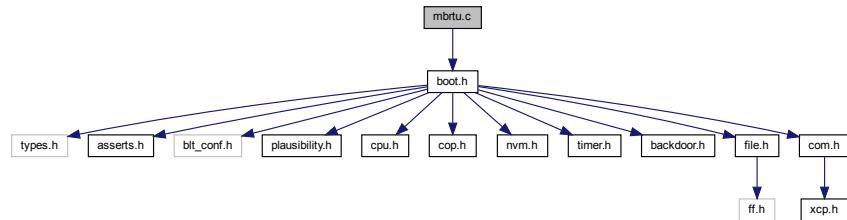
7.184.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.185 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_STM32L1/mbrtu.c:
```



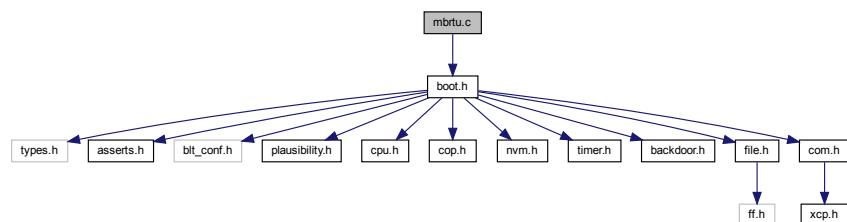
7.185.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.186 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_S32K14/mbrtu.c:
```



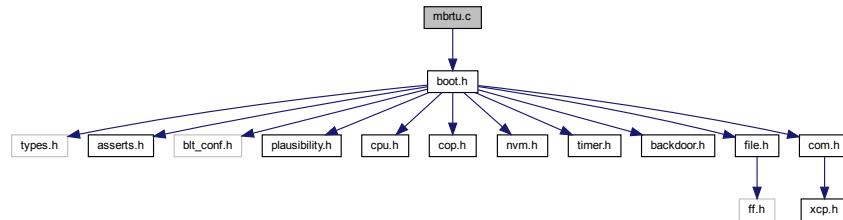
7.186.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.187 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_STM32F3/mbrtu.c:
```



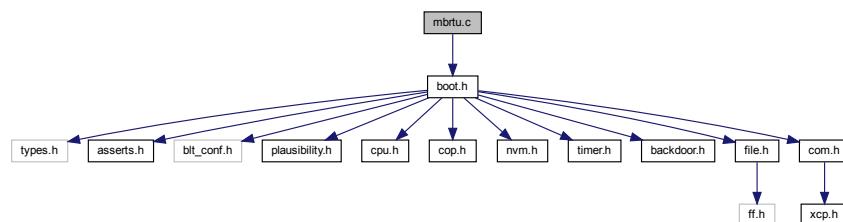
7.187.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.188 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_STM32F4/mbrtu.c:
```



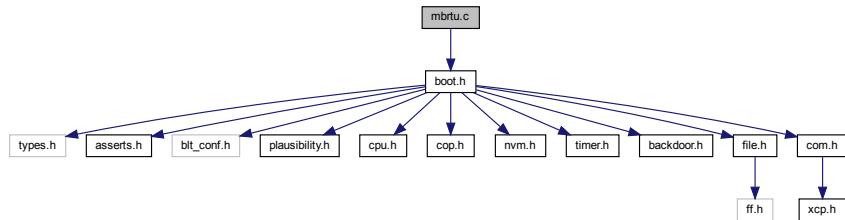
7.188.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.189 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_STM32G4/mbrtu.c:
```



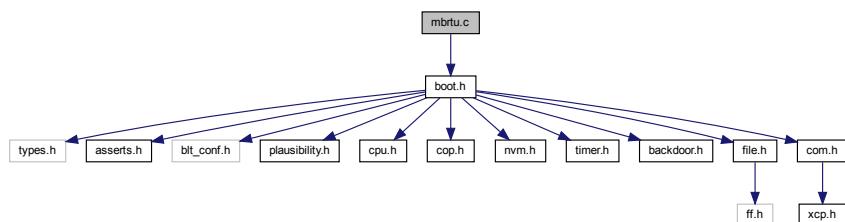
7.189.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.190 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_STM32L4/mbrtu.c:
```



7.190.1 Detailed Description

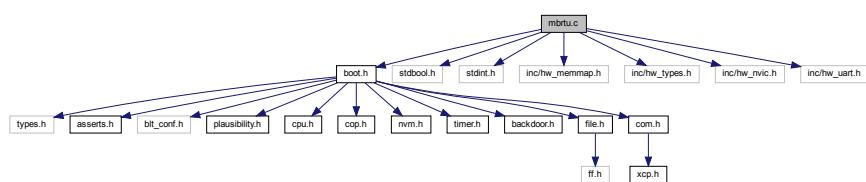
Bootloader Modbus RTU communication interface source file.

7.191 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_nvic.h"
#include "inc/hw_uart.h"
```

Include dependency graph for ARMCM4_TM4C/mbrtu.c:



7.191.1 Detailed Description

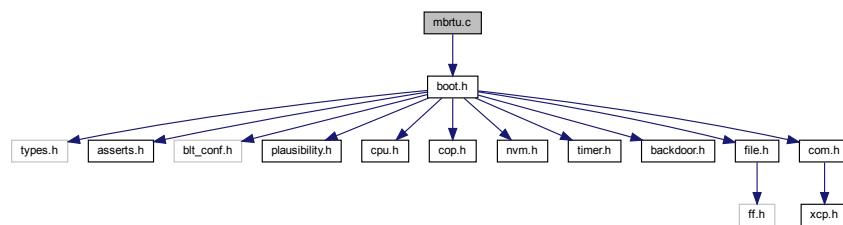
Bootloader Modbus RTU communication interface source file.

7.192 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
```

Include dependency graph for ARMCM4_XMC4/mbrtu.c:



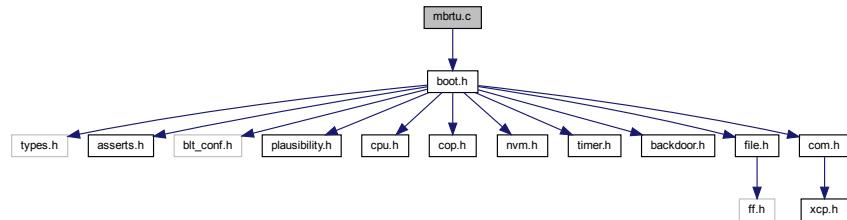
7.192.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.193 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM7_STM32F7/mbrtu.c:
```



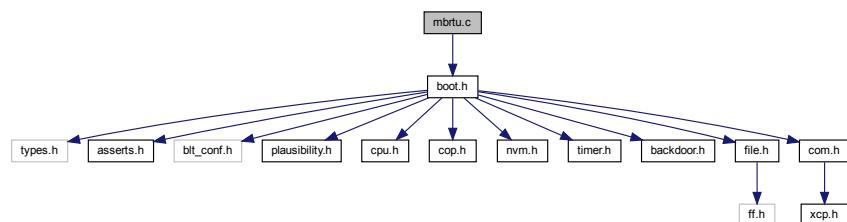
7.193.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.194 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM7_STM32H7/mbrtu.c:
```



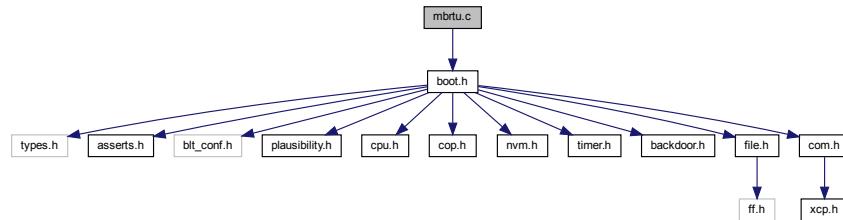
7.194.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.195 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for HCS12/mbrtu.c:
```



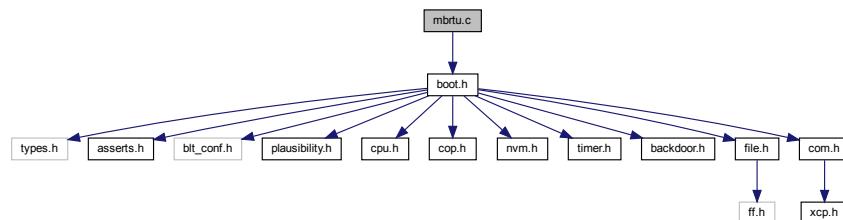
7.195.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.196 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for TRICORE_TC2/mbrtu.c:
```



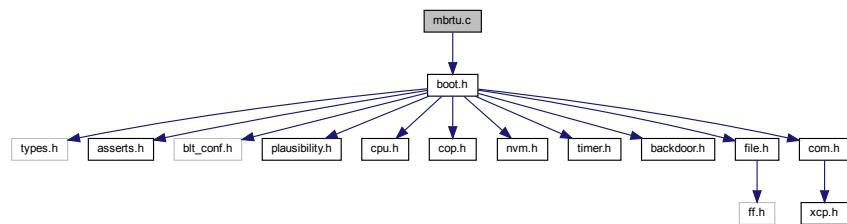
7.196.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.197 mbrtu.c File Reference

Bootloader Modbus RTU communication interface source file.

```
#include "boot.h"
Include dependency graph for TRICORE_TC3/mbrtu.c:
```



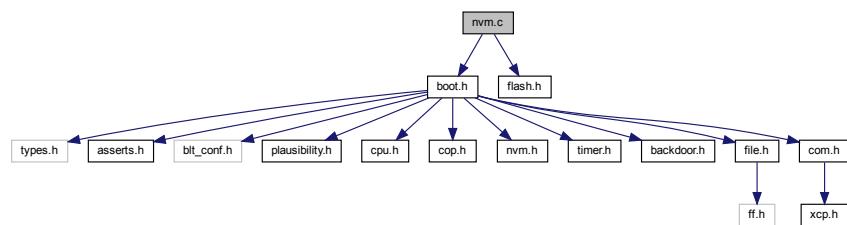
7.197.1 Detailed Description

Bootloader Modbus RTU communication interface source file.

7.198 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for _template/nvm.c:
```



Functions

- `void NvmInit (void)`
Initializes the NVM driver.
- `blt_bool NvmWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Programs the non-volatile memory.
- `blt_bool NvmErase (blt_addr addr, blt_int32u len)`
Erases the non-volatile memory.
- `blt_bool NvmVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_addr Nvm GetUserProgBaseAddress (void)`
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- `blt_bool NvmDone (void)`
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.198.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.198.2 Function Documentation

7.198.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

`BLT_TRUE` if successful, `BLT_FALSE` otherwise.

Referenced by [XcpCmdProgram\(\)](#).

7.198.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [XcpCmdProgramClear\(\)](#).

7.198.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.198.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

Referenced by [BootInit\(\)](#), [FileTask\(\)](#), and [XcpCmdConnect\(\)](#).

7.198.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [CpuStartUserProgram\(\)](#).

7.198.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

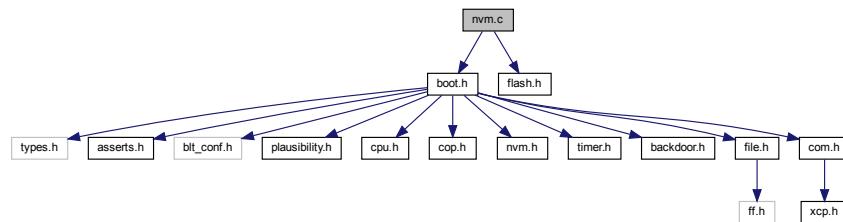
BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [XcpCmdProgram\(\)](#), and [XcpCmdProgramMax\(\)](#).

7.199 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM0_S32K11/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.199.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.199.2 Function Documentation

7.199.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.199.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.199.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.199.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.199.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.199.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

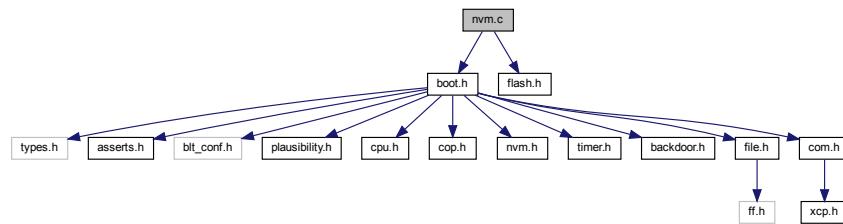
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.200 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM0_STM32C0/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.200.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.200.2 Function Documentation

7.200.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.200.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.200.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.200.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.200.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.200.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

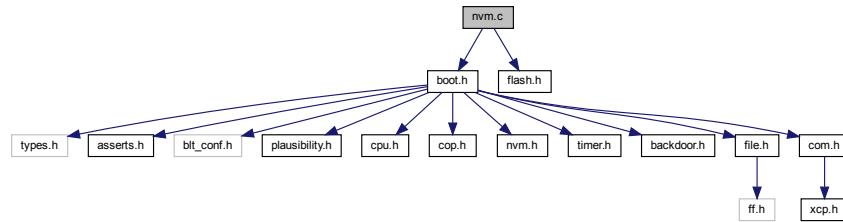
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.201 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM0_STM32F0/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.201.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.201.2 Function Documentation

7.201.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.201.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.201.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.201.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.201.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.201.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

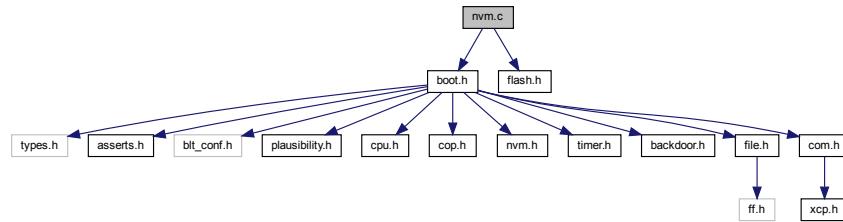
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.202 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM0_STM32G0/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.202.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.202.2 Function Documentation

7.202.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.202.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.202.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.202.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.202.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.202.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

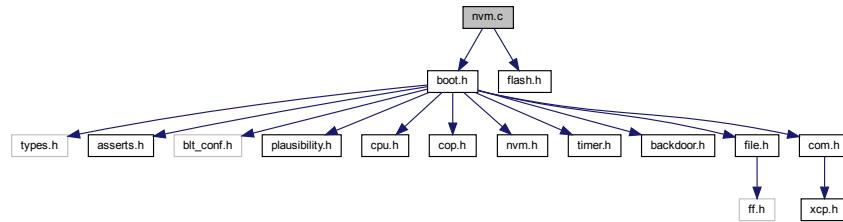
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.203 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM0_STM32L0/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.203.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.203.2 Function Documentation

7.203.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.203.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.203.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.203.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.203.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.203.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

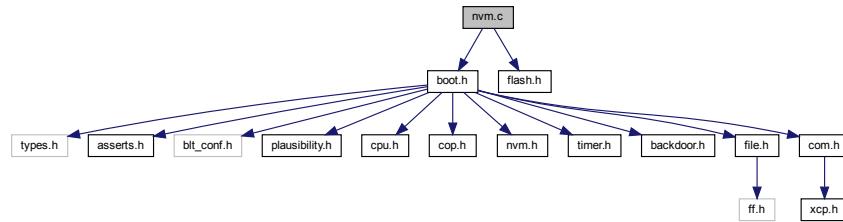
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.204 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM0_XMC1/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.204.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.204.2 Function Documentation

7.204.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.204.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.204.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.204.2.4 NvmInit()

```
void NvmInit (
    void  )
```

Initializes the NVM driver.

Returns

none.

7.204.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void  )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.204.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

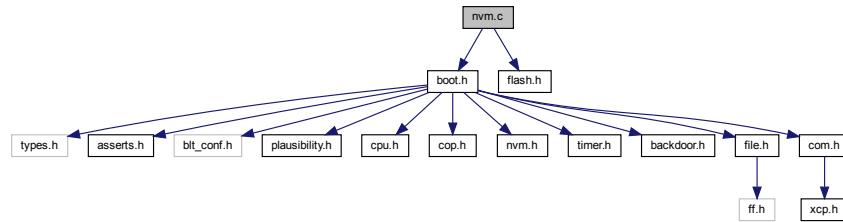
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.205 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM33_STM32H5/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.205.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.205.2 Function Documentation

7.205.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.205.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.205.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.205.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.205.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.205.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

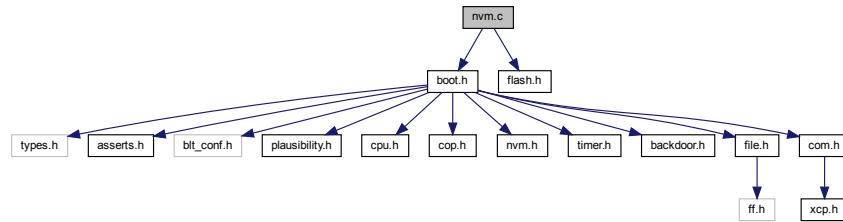
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.206 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM33_STM32L5/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.206.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.206.2 Function Documentation

7.206.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.206.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.206.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.206.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.206.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.206.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

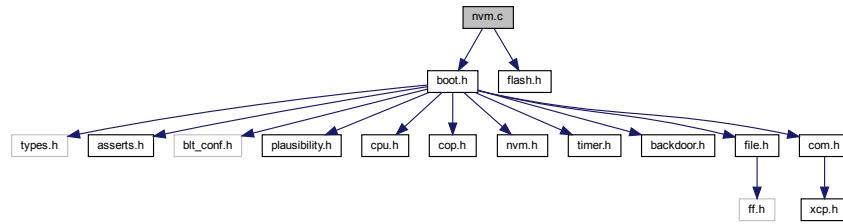
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.207 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM33_STM32U5/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.207.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.207.2 Function Documentation

7.207.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.207.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.207.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.207.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.207.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.207.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

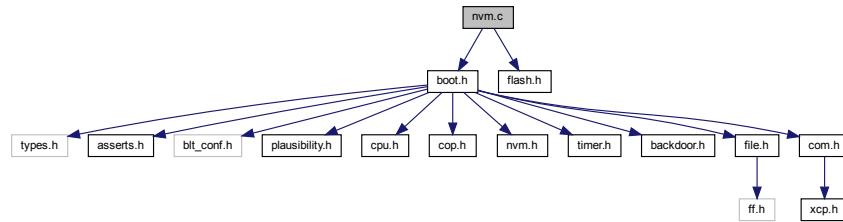
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.208 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM3_EFM32/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.208.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.208.2 Function Documentation

7.208.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.208.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.208.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.208.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.208.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.208.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

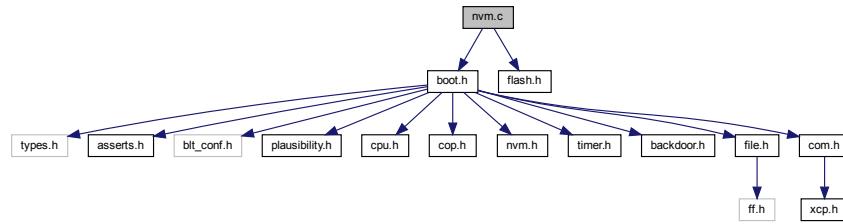
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.209 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM3_LM3S/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.209.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.209.2 Function Documentation

7.209.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.209.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.209.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.209.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.209.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.209.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

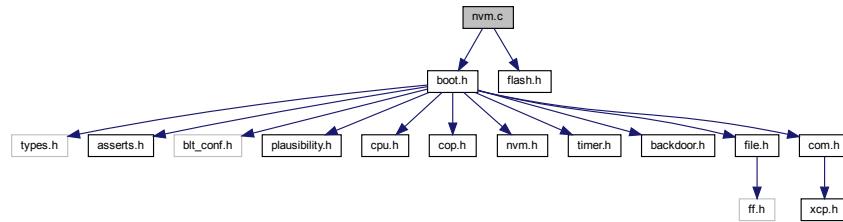
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.210 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM3_STM32F1/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.210.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.210.2 Function Documentation

7.210.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.210.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.210.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.210.2.4 NvmInit()

```
void NvmInit (
    void  )
```

Initializes the NVM driver.

Returns

none.

7.210.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void  )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.210.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

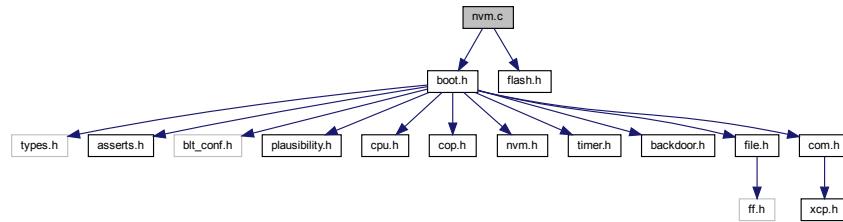
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.211 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM3_STM32F2/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.211.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.211.2 Function Documentation

7.211.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.211.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.211.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.211.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.211.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.211.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

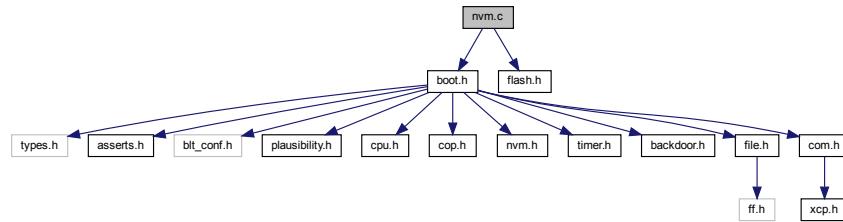
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.212 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM3_STM32L1/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.212.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.212.2 Function Documentation

7.212.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.212.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.212.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.212.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.212.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.212.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

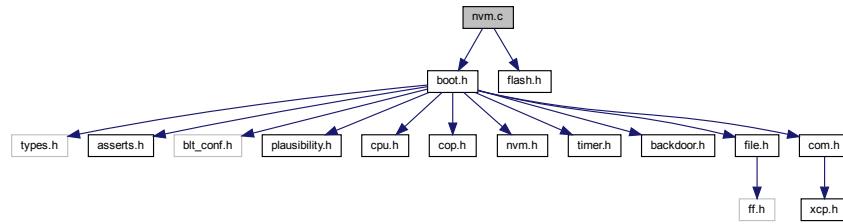
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.213 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM4_S32K14/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.213.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.213.2 Function Documentation

7.213.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.213.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.213.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.213.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.213.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.213.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

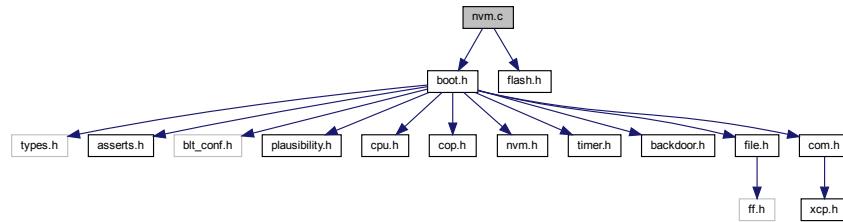
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.214 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM4_STM32F3/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.214.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.214.2 Function Documentation

7.214.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.214.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.214.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.214.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.214.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.214.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

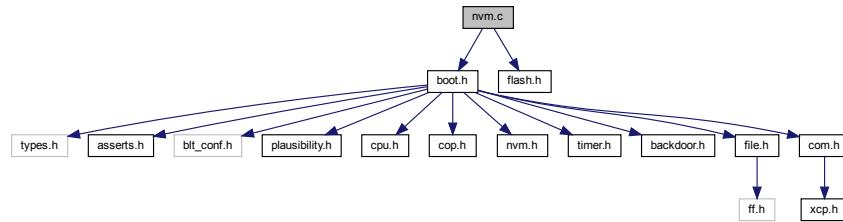
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.215 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM4_STM32F4/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.215.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.215.2 Function Documentation

7.215.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.215.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.215.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.215.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.215.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.215.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

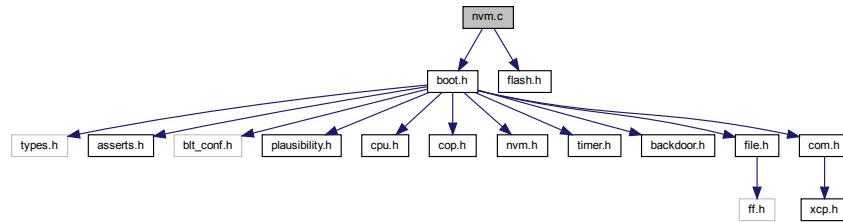
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.216 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM4_STM32G4/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.216.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.216.2 Function Documentation

7.216.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.216.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.216.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.216.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.216.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.216.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

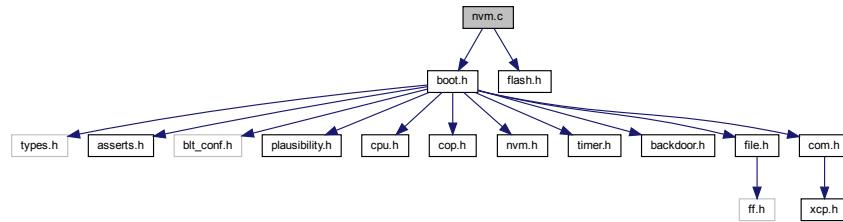
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.217 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM4_STM32L4/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.217.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.217.2 Function Documentation

7.217.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.217.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.217.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.217.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.217.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.217.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

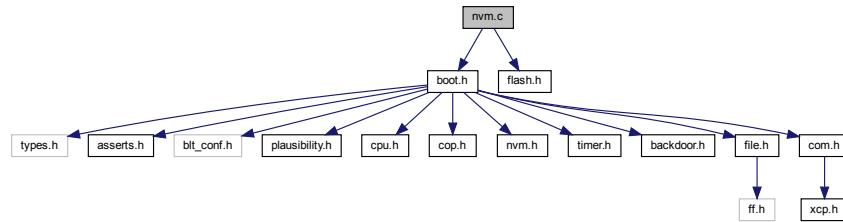
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.218 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM4_TM4C/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.218.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.218.2 Function Documentation

7.218.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.218.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.218.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.218.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.218.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.218.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

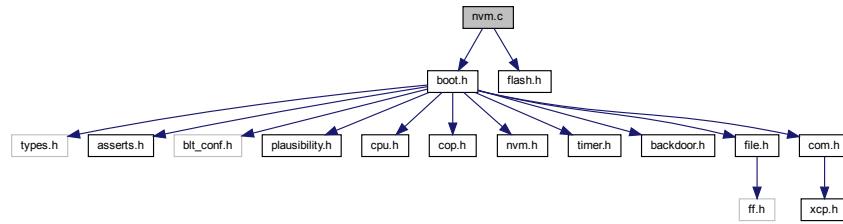
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.219 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM4_XMC4/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.219.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.219.2 Function Documentation

7.219.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.219.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.219.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.219.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.219.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.219.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

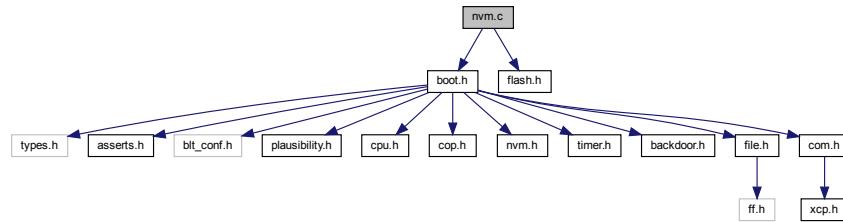
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.220 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM7_STM32F7/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.220.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.220.2 Function Documentation

7.220.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.220.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.220.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.220.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.220.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.220.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

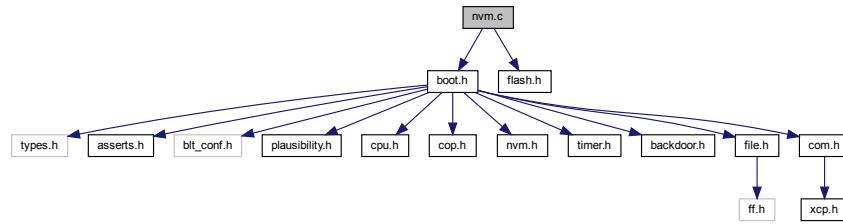
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.221 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for ARMCM7_STM32H7/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.221.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.221.2 Function Documentation

7.221.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.221.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.221.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.221.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.221.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.221.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

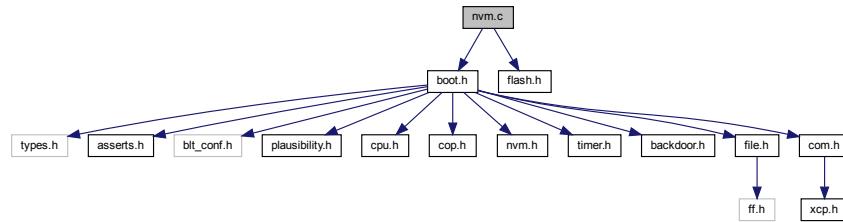
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.222 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for HCS12/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.222.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.222.2 Function Documentation

7.222.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.222.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.222.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.222.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.222.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.222.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

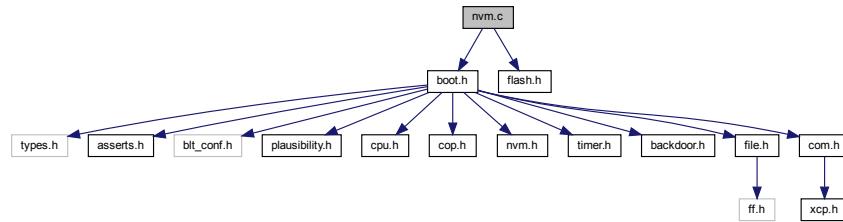
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.223 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for TRICORE_TC2/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.223.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.223.2 Function Documentation

7.223.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.223.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.223.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.223.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.223.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.223.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

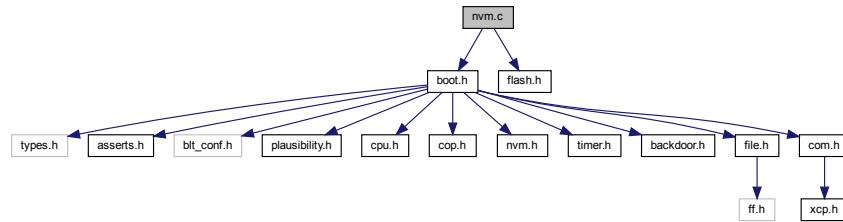
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.224 nvm.c File Reference

Bootloader non-volatile memory driver source file.

```
#include "boot.h"
#include "flash.h"
Include dependency graph for TRICORE_TC3/nvm.c:
```



Functions

- void [NvmlInit](#) (void)
Initializes the NVM driver.
- [blt_bool NvmWrite](#) ([blt_addr](#) addr, [blt_int32u](#) len, [blt_int8u](#) *data)
Programs the non-volatile memory.
- [blt_bool NvmErase](#) ([blt_addr](#) addr, [blt_int32u](#) len)
Erases the non-volatile memory.
- [blt_bool NvmVerifyChecksum](#) (void)
Verifies the checksum, which indicates that a valid user program is present and can be started.
- [blt_addr Nvm GetUserProgBaseAddress](#) (void)
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- [blt_bool NvmDone](#) (void)
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.224.1 Detailed Description

Bootloader non-volatile memory driver source file.

7.224.2 Function Documentation

7.224.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.224.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.224.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.224.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

7.224.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.224.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

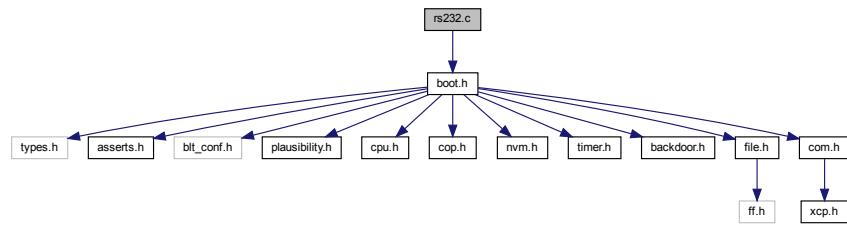
Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.225 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for _template/rs232.c:
```



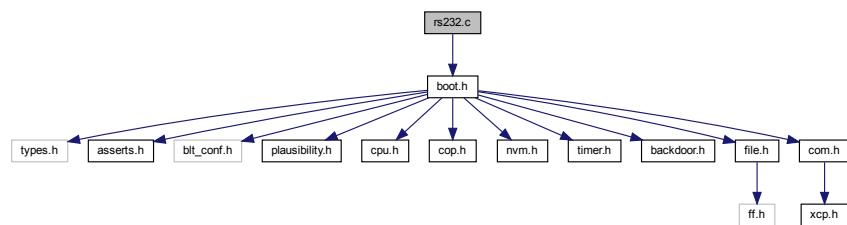
7.225.1 Detailed Description

Bootloader RS232 communication interface source file.

7.226 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_S32K11/rs232.c:
```



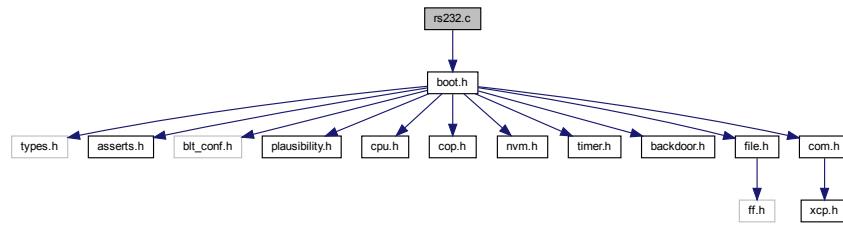
7.226.1 Detailed Description

Bootloader RS232 communication interface source file.

7.227 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_STM32C0/rs232.c:
```



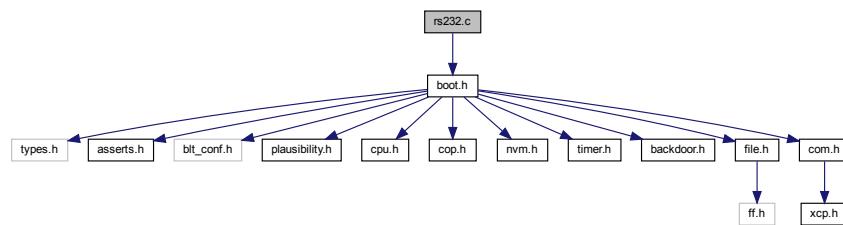
7.227.1 Detailed Description

Bootloader RS232 communication interface source file.

7.228 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_STM32F0/rs232.c:
```



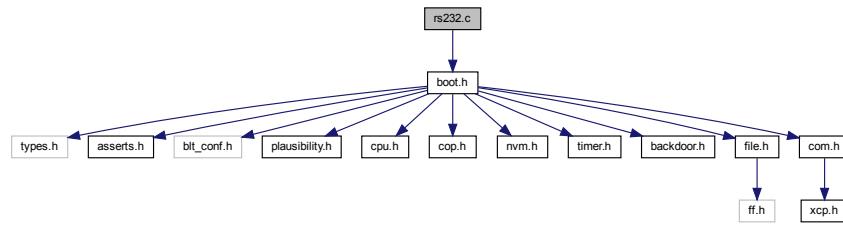
7.228.1 Detailed Description

Bootloader RS232 communication interface source file.

7.229 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"  
Include dependency graph for ARMCM0_STM32G0/rs232.c:
```



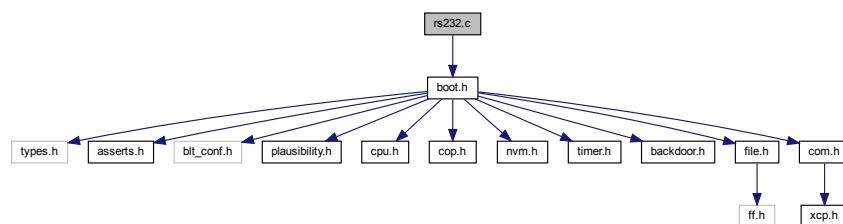
7.229.1 Detailed Description

Bootloader RS232 communication interface source file.

7.230 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"  
Include dependency graph for ARMCM0_STM32L0/rs232.c:
```



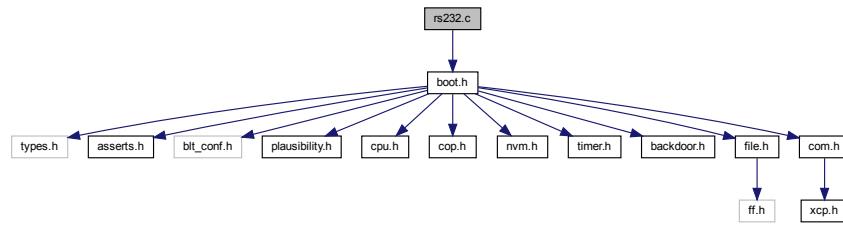
7.230.1 Detailed Description

Bootloader RS232 communication interface source file.

7.231 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM0_XMC1/rs232.c:
```



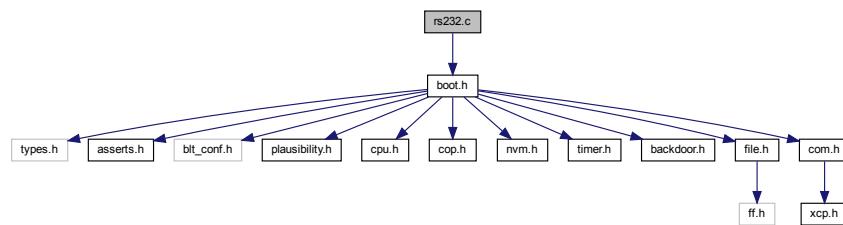
7.231.1 Detailed Description

Bootloader RS232 communication interface source file.

7.232 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM33_STM32H5/rs232.c:
```



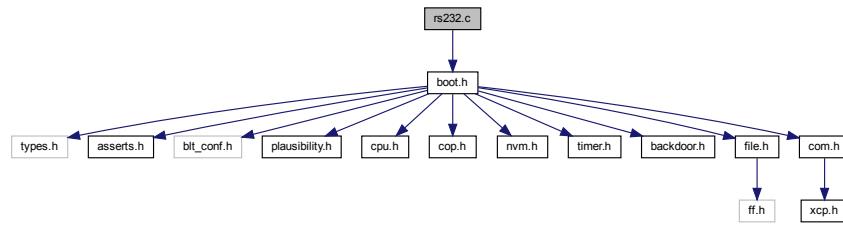
7.232.1 Detailed Description

Bootloader RS232 communication interface source file.

7.233 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"  
Include dependency graph for ARMCM33_STM32L5/rs232.c:
```



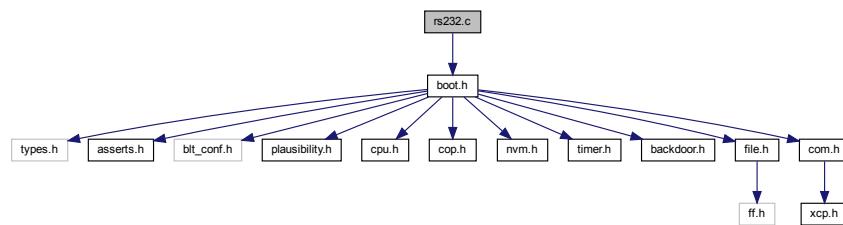
7.233.1 Detailed Description

Bootloader RS232 communication interface source file.

7.234 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"  
Include dependency graph for ARMCM33_STM32U5/rs232.c:
```



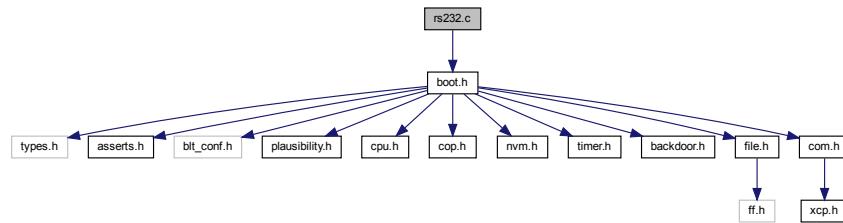
7.234.1 Detailed Description

Bootloader RS232 communication interface source file.

7.235 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_EFM32/rs232.c:
```



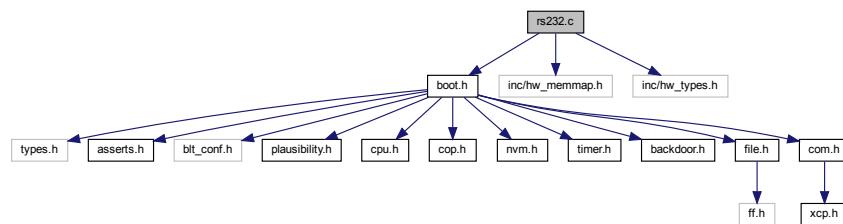
7.235.1 Detailed Description

Bootloader RS232 communication interface source file.

7.236 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
Include dependency graph for ARMCM3_LM3S/rs232.c:
```



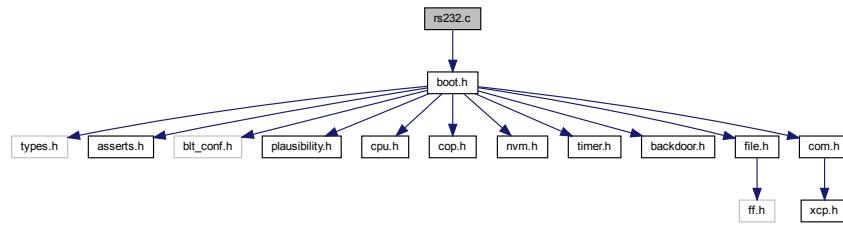
7.236.1 Detailed Description

Bootloader RS232 communication interface source file.

7.237 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_STM32F1/rs232.c:
```



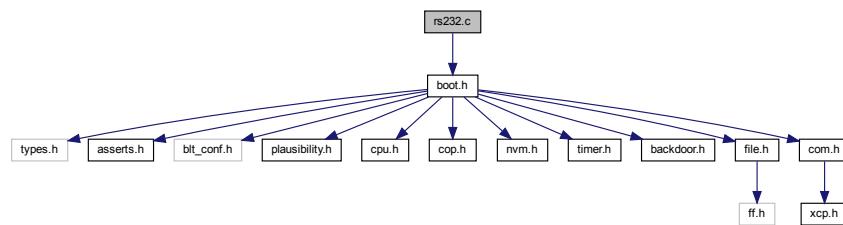
7.237.1 Detailed Description

Bootloader RS232 communication interface source file.

7.238 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_STM32F2/rs232.c:
```



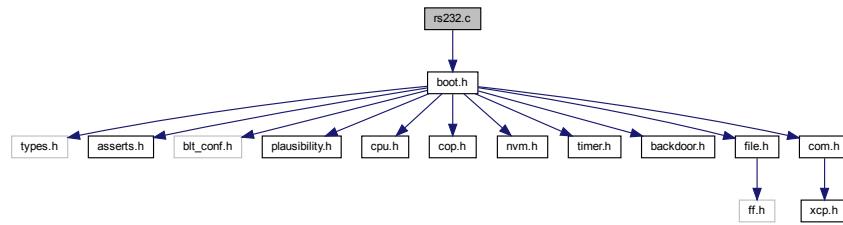
7.238.1 Detailed Description

Bootloader RS232 communication interface source file.

7.239 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM3_STM32L1/rs232.c:
```



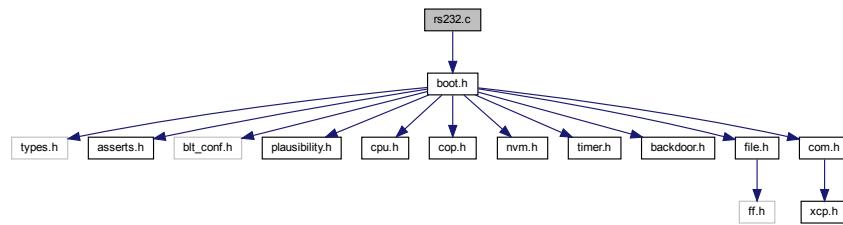
7.239.1 Detailed Description

Bootloader RS232 communication interface source file.

7.240 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_S32K14/rs232.c:
```



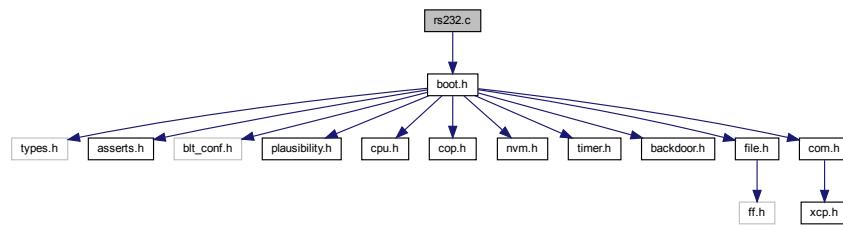
7.240.1 Detailed Description

Bootloader RS232 communication interface source file.

7.241 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_STM32F3/rs232.c:
```



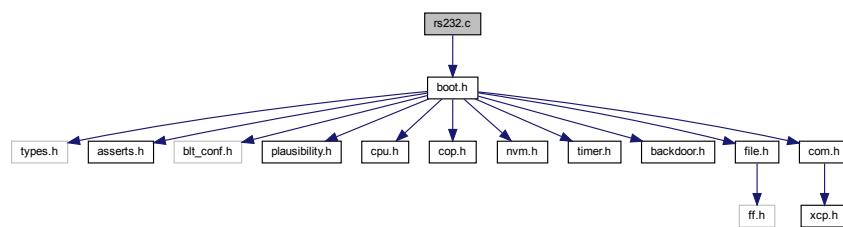
7.241.1 Detailed Description

Bootloader RS232 communication interface source file.

7.242 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_STM32F4/rs232.c:
```



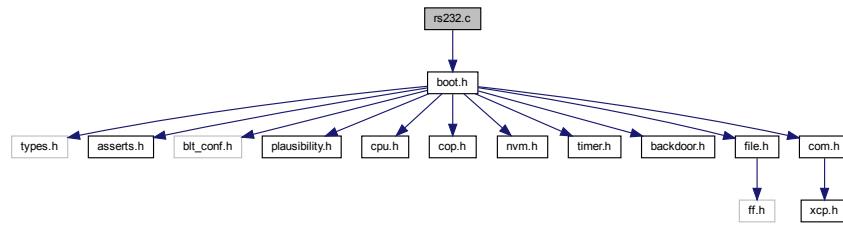
7.242.1 Detailed Description

Bootloader RS232 communication interface source file.

7.243 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_STM32G4/rs232.c:
```



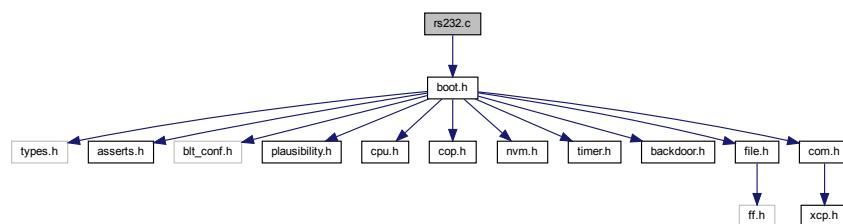
7.243.1 Detailed Description

Bootloader RS232 communication interface source file.

7.244 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_STM32L4/rs232.c:
```



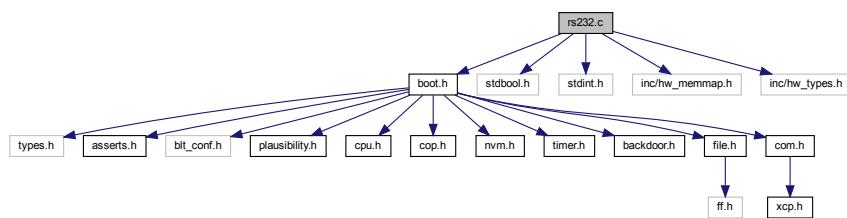
7.244.1 Detailed Description

Bootloader RS232 communication interface source file.

7.245 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
Include dependency graph for ARMCM4_TM4C/rs232.c:
```



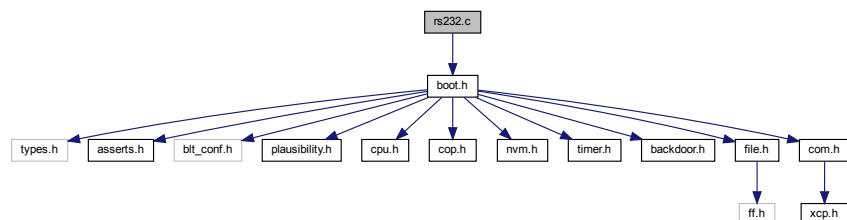
7.245.1 Detailed Description

Bootloader RS232 communication interface source file.

7.246 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM4_XMC4/rs232.c:
```



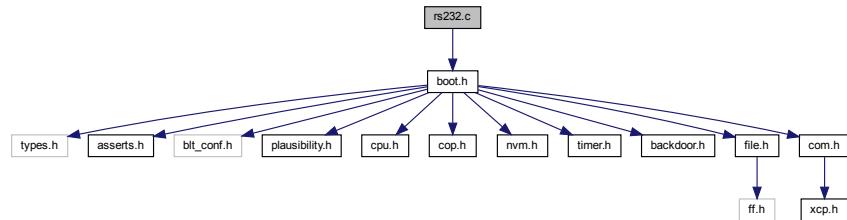
7.246.1 Detailed Description

Bootloader RS232 communication interface source file.

7.247 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM7_STM32F7/rs232.c:
```



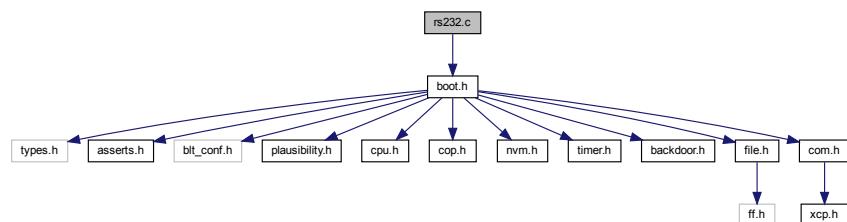
7.247.1 Detailed Description

Bootloader RS232 communication interface source file.

7.248 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for ARMCM7_STM32H7/rs232.c:
```



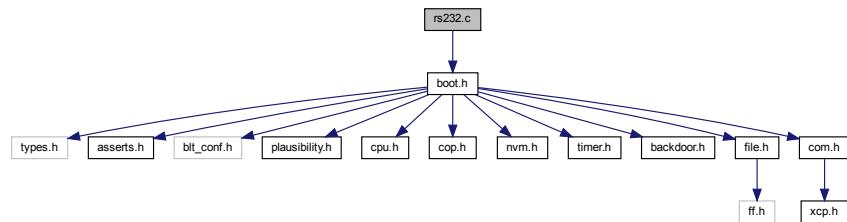
7.248.1 Detailed Description

Bootloader RS232 communication interface source file.

7.249 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for HCS12/rs232.c:
```



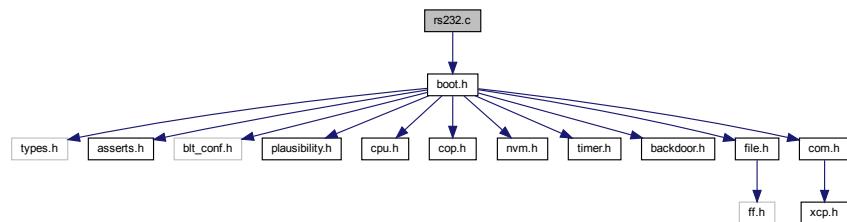
7.249.1 Detailed Description

Bootloader RS232 communication interface source file.

7.250 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for TRICORE_TC2/rs232.c:
```



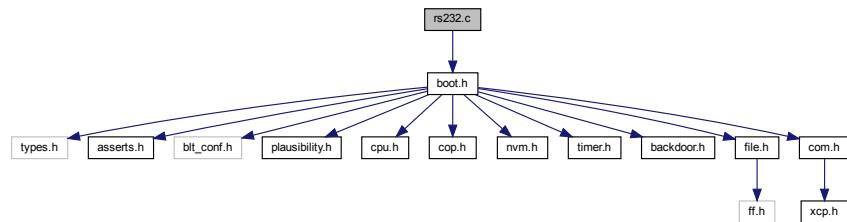
7.250.1 Detailed Description

Bootloader RS232 communication interface source file.

7.251 rs232.c File Reference

Bootloader RS232 communication interface source file.

```
#include "boot.h"
Include dependency graph for TRICORE_TC3/rs232.c:
```



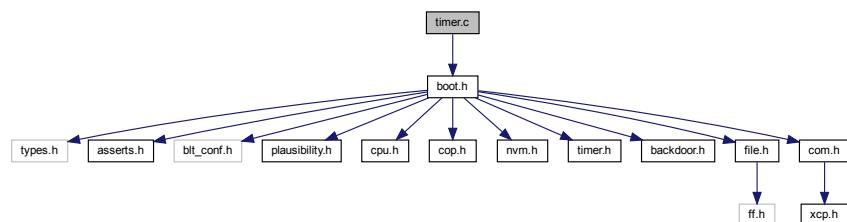
7.251.1 Detailed Description

Bootloader RS232 communication interface source file.

7.252 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
Include dependency graph for _template/timer.c:
```



Functions

- void [TimerInit](#) (void)
Initializes the polling based millisecond timer driver.
- void [TimerReset](#) (void)
Reset the timer by placing the timer back into it's default reset configuration.
- void [TimerUpdate](#) (void)
Updates the millisecond timer.
- [blt_int32u TimerGet](#) (void)
Obtains the counter value of the millisecond timer.

Variables

- static [blt_int32u millisecond_counter](#)
Local variable for storing the number of milliseconds that have elapsed since startup.

7.252.1 Detailed Description

Bootloader timer driver source file.

7.252.2 Function Documentation

7.252.2.1 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [BackDoorCheck\(\)](#), [CanDisabledModeEnter\(\)](#), [CanDisabledModeExit\(\)](#), [CanFreezeModeEnter\(\)](#), [CanFreezeModeExit\(\)](#), [CanInit\(\)](#), [CanTransmitPacket\(\)](#), [FlashEraseSectors\(\)](#), [FlashWriteBlock\(\)](#), [NetInit\(\)](#), and [NetServerTask\(\)](#).

7.252.2.2 TimerInit()

```
void TimerInit (
    void  )
```

Initializes the polling based millisecond timer driver.

Returns

none.

Referenced by [BootInit\(\)](#).

7.252.2.3 TimerReset()

```
void TimerReset (
    void  )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

Referenced by [TimerInit\(\)](#).

7.252.2.4 TimerUpdate()

```
void TimerUpdate (
    void  )
```

Updates the millisecond timer.

Returns

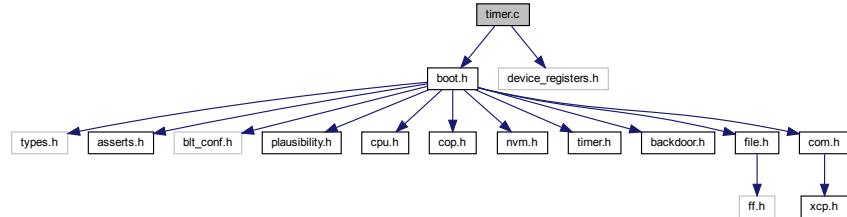
none.

Referenced by [BootTask\(\)](#), and [TimerGet\(\)](#).

7.253 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "device_registers.h"
Include dependency graph for ARMCM0_S32K11/timer.c:
```



Functions

- void **TimerInit** (void)
Initializes the polling based millisecond timer driver.
- void **TimerReset** (void)
Reset the timer by placing the timer back into it's default reset configuration.
- void **TimerUpdate** (void)
Updates the millisecond timer.
- **blt_int32u TimerGet** (void)
Obtains the counter value of the millisecond timer.

Variables

- **blt_int32u SystemCoreClock**
The system clock frequency supplied to the SysTick timer and the processor core clock.
- static **blt_int32u millisecond_counter**
Local variable for storing the number of milliseconds that have elapsed since startup.
- static **blt_int32u free_running_counter_last**
Buffer for storing the last value of the free running counter.
- static **blt_int32u counts_per_millisecond**
Stores the number of counts of the free running counter that equals one millisecond.

7.253.1 Detailed Description

Bootloader timer driver source file.

7.253.2 Function Documentation

7.253.2.1 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

7.253.2.2 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Ideally a 100 kHz free running counter is used as the foundation for the timer modules, as this gives 10us ticks that can be reused by other modules. The S32K11 timers unfortunately do not offer a flexible prescaler for their timers to realize such a 100 kHz free running counter. For this reason, the SysTick counter is used instead. Its 24-bit free running down-counter runs at the system speed.

Returns

none.

7.253.2.3 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

Referenced by [TimerInit\(\)](#).

7.253.2.4 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

none.

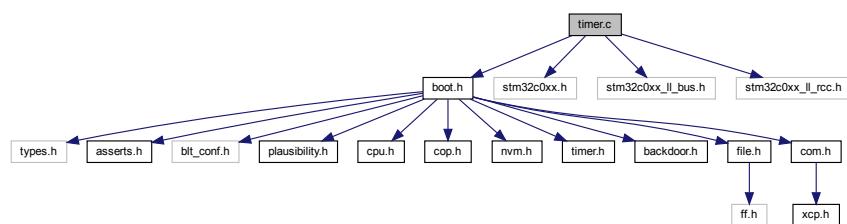
Referenced by [TimerGet\(\)](#).

7.254 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32c0xx.h"
#include "stm32c0xx_ll_bus.h"
#include "stm32c0xx_ll_rcc.h"
```

Include dependency graph for ARMCM0_STM32C0/timer.c:



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into its default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- static `blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- static `blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.254.1 Detailed Description

Bootloader timer driver source file.

7.254.2 Function Documentation

7.254.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.254.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.254.2.3 TimerInit()

```
void TimerInit (
    void  )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.254.2.4 TimerReset()

```
void TimerReset (
    void  )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.254.2.5 TimerUpdate()

```
void TimerUpdate (
    void  )
```

Updates the millisecond timer.

Returns

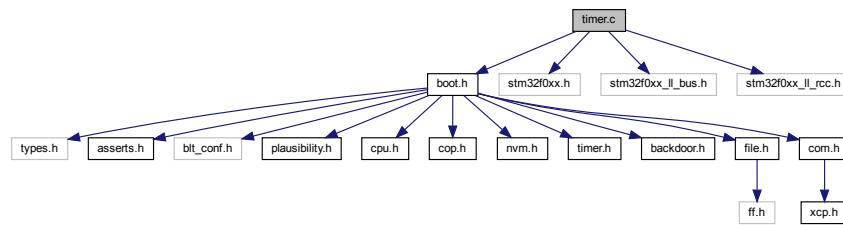
none.

Referenced by [TimerGet\(\)](#).

7.255 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32f0xx.h"
#include "stm32f0xx_ll_bus.h"
#include "stm32f0xx_ll_rcc.h"
Include dependency graph for ARMCM0_STM32F0/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into it's default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- `static blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- `static blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.255.1 Detailed Description

Bootloader timer driver source file.

7.255.2 Function Documentation

7.255.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.255.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.255.2.3 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.255.2.4 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.255.2.5 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

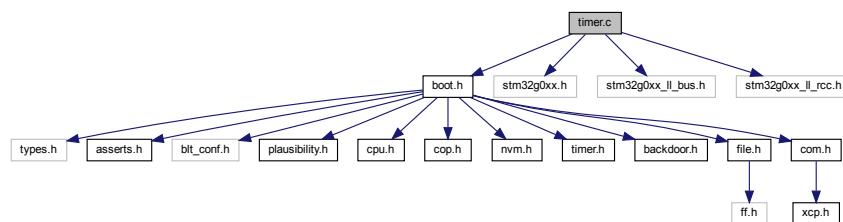
none.

Referenced by [TimerGet\(\)](#).

7.256 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32g0xx.h"
#include "stm32g0xx_ll_bus.h"
#include "stm32g0xx_ll_rcc.h"
Include dependency graph for ARMCM0_STM32G0/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into its default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- `static blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- `static blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.256.1 Detailed Description

Bootloader timer driver source file.

7.256.2 Function Documentation

7.256.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.256.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.256.2.3 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.256.2.4 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.256.2.5 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

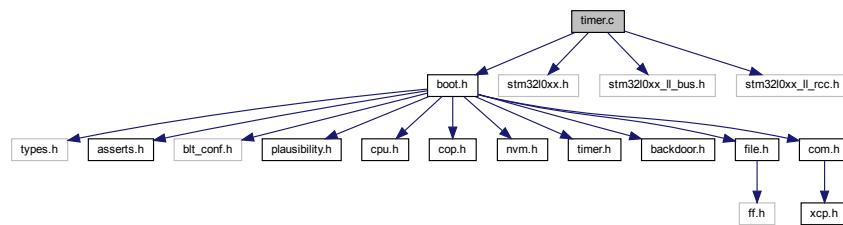
none.

Referenced by [TimerGet\(\)](#).

7.257 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32l0xx.h"
#include "stm32l0xx_ll_bus.h"
#include "stm32l0xx_ll_rcc.h"
Include dependency graph for ARMCM0_STM32L0/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into its default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- static `blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- static `blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.257.1 Detailed Description

Bootloader timer driver source file.

7.257.2 Function Documentation

7.257.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.257.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.257.2.3 TimerInit()

```
void TimerInit (
    void  )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.257.2.4 TimerReset()

```
void TimerReset (
    void  )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.257.2.5 TimerUpdate()

```
void TimerUpdate (
    void  )
```

Updates the millisecond timer.

Returns

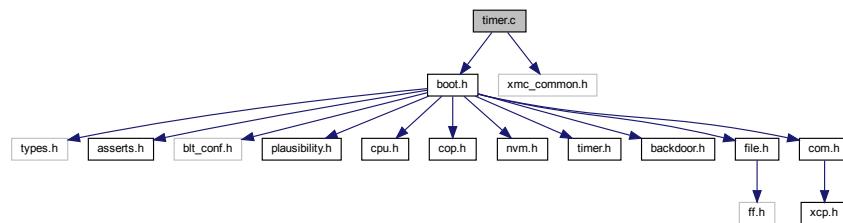
none.

Referenced by [TimerGet\(\)](#).

7.258 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "xmc_common.h"
Include dependency graph for ARMCM0_XMC1/timer.c:
```



Functions

- void [TimerInit](#) (void)
Initializes the polling based millisecond timer driver.
- void [TimerReset](#) (void)
Reset the timer by placing the timer back into it's default reset configuration.
- void [TimerUpdate](#) (void)
Updates the millisecond timer.
- [blt_int32u TimerGet](#) (void)
Obtains the counter value of the millisecond timer.

Variables

- static [blt_int32u millisecond_counter](#)
Local variable for storing the number of milliseconds that have elapsed since startup.
- static [blt_int32u free_running_counter_last](#)
Buffer for storing the last value of the free running counter.
- static [blt_int32u counts_per_millisecond](#)
Stores the number of counts of the free running counter that equals one millisecond.

7.258.1 Detailed Description

Bootloader timer driver source file.

7.258.2 Function Documentation

7.258.2.1 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

7.258.2.2 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Ideally a 100 kHz free running counter is used as the foundation for the timer modules, as this gives 10us ticks that can be reused by other modules. The XMC1 timers unfortunately do not offer a flexible prescaler for their timers to realize such a 100 kHz free running counter. For this reason, the SysTick counter is used instead. Its 24-bit free running down-counter runs at the system speed.

Returns

none.

7.258.2.3 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

Referenced by [TimerInit\(\)](#).

7.258.2.4 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

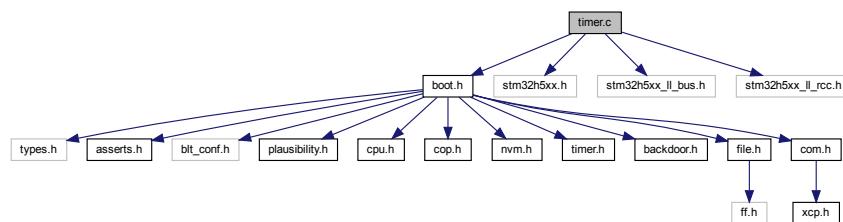
none.

Referenced by [TimerGet\(\)](#).

7.259 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32h5xx.h"
#include "stm32h5xx_ll_bus.h"
#include "stm32h5xx_ll_rcc.h"
Include dependency graph for ARMCM33_STM32H5/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into its default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- static `blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- static `blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.259.1 Detailed Description

Bootloader timer driver source file.

7.259.2 Function Documentation

7.259.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.259.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.259.2.3 TimerInit()

```
void TimerInit (
    void  )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.259.2.4 TimerReset()

```
void TimerReset (
    void  )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.259.2.5 TimerUpdate()

```
void TimerUpdate (
    void  )
```

Updates the millisecond timer.

Returns

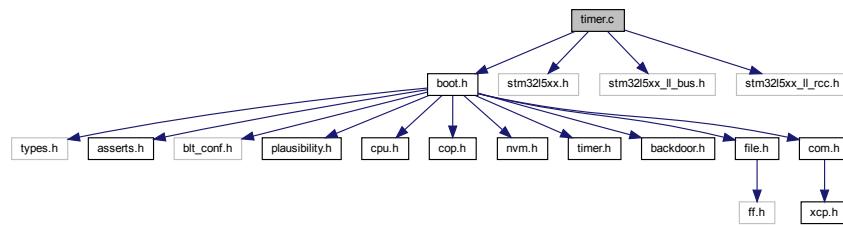
none.

Referenced by [TimerGet\(\)](#).

7.260 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32l5xx.h"
#include "stm32l5xx_ll_bus.h"
#include "stm32l5xx_ll_rcc.h"
Include dependency graph for ARMCM33_STM32L5/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into it's default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- `static blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- `static blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.260.1 Detailed Description

Bootloader timer driver source file.

7.260.2 Function Documentation

7.260.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.260.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.260.2.3 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.260.2.4 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.260.2.5 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

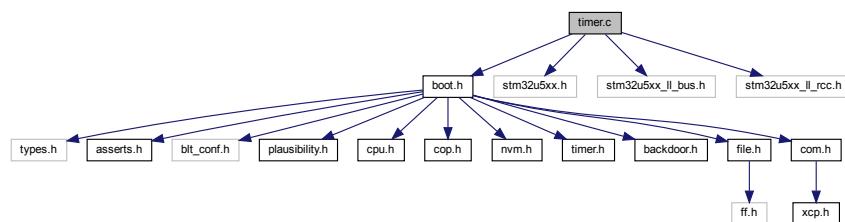
none.

Referenced by [TimerGet\(\)](#).

7.261 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32u5xx.h"
#include "stm32u5xx_ll_bus.h"
#include "stm32u5xx_ll_rcc.h"
Include dependency graph for ARMCM33_STM32U5/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into its default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- `static blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- `static blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.261.1 Detailed Description

Bootloader timer driver source file.

7.261.2 Function Documentation

7.261.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.261.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.261.2.3 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.261.2.4 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.261.2.5 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

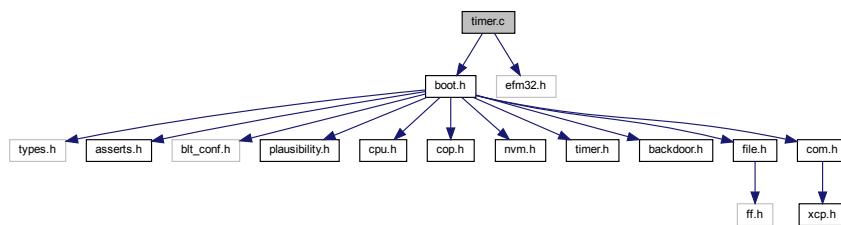
none.

Referenced by [TimerGet\(\)](#).

7.262 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "efm32.h"
Include dependency graph for ARMCM3_EFM32/timer.c:
```



Functions

- void [TimerInit](#) (void)
Initializes the polling based millisecond timer driver.
- void [TimerReset](#) (void)
Reset the timer by placing the timer back into it's default reset configuration.
- void [TimerUpdate](#) (void)
Updates the millisecond timer.
- [blt_int32u TimerGet](#) (void)
Obtains the counter value of the millisecond timer.

Variables

- static [blt_int32u millisecond_counter](#)
Local variable for storing the number of milliseconds that have elapsed since startup.
- static [blt_int32u free_running_counter_last](#)
Buffer for storing the last value of the free running counter.
- static [blt_int32u counts_per_millisecond](#)
Stores the number of counts of the free running counter that equals one millisecond.

7.262.1 Detailed Description

Bootloader timer driver source file.

7.262.2 Function Documentation

7.262.2.1 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

7.262.2.2 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Returns

none.

7.262.2.3 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

Referenced by [TimerInit\(\)](#).

7.262.2.4 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

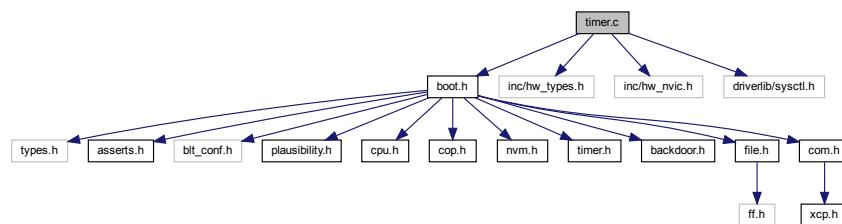
none.

Referenced by [TimerGet\(\)](#).

7.263 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "inc/hw_types.h"
#include "inc/hw_nvic.h"
#include "driverlib/sysctl.h"
Include dependency graph for ARMCM3_LM3S/timer.c:
```



Functions

- void [TimerInit](#) (void)
Initializes the polling based millisecond timer driver.
- void [TimerReset](#) (void)
Reset the timer by placing the timer back into it's default reset configuration.
- void [TimerUpdate](#) (void)
Updates the millisecond timer.
- [blt_int32u](#) [TimerGet](#) (void)
Obtains the counter value of the millisecond timer.

Variables

- static [blt_int32u](#) **millisecond_counter**
Local variable for storing the number of milliseconds that have elapsed since startup.
- static [blt_int32u](#) **free_running_counter_last**
Buffer for storing the last value of the free running counter.
- static [blt_int32u](#) **counts_per_millisecond**
Stores the number of counts of the free running counter that equals one millisecond.

7.263.1 Detailed Description

Bootloader timer driver source file.

7.263.2 Function Documentation

7.263.2.1 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

7.263.2.2 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Returns

none.

7.263.2.3 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

Referenced by [TimerInit\(\)](#).

7.263.2.4 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

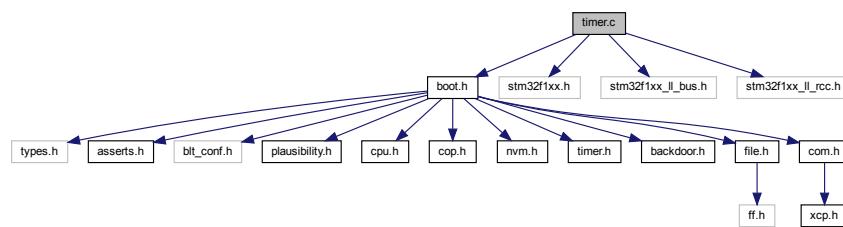
none.

Referenced by [TimerGet\(\)](#).

7.264 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32f1xx.h"
#include "stm32f1xx_ll_bus.h"
#include "stm32f1xx_ll_rcc.h"
Include dependency graph for ARMCM3_STM32F1/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into its default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- static `blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- static `blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.264.1 Detailed Description

Bootloader timer driver source file.

7.264.2 Function Documentation

7.264.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.264.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.264.2.3 TimerInit()

```
void TimerInit (
    void  )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.264.2.4 TimerReset()

```
void TimerReset (
    void  )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.264.2.5 TimerUpdate()

```
void TimerUpdate (
    void  )
```

Updates the millisecond timer.

Returns

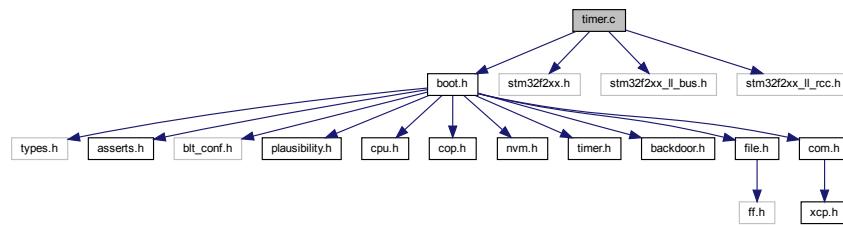
none.

Referenced by [TimerGet\(\)](#).

7.265 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32f2xx.h"
#include "stm32f2xx_ll_bus.h"
#include "stm32f2xx_ll_rcc.h"
Include dependency graph for ARCMC3_STM32F2/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into it's default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- `static blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- `static blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.265.1 Detailed Description

Bootloader timer driver source file.

7.265.2 Function Documentation

7.265.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.265.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.265.2.3 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.265.2.4 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.265.2.5 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

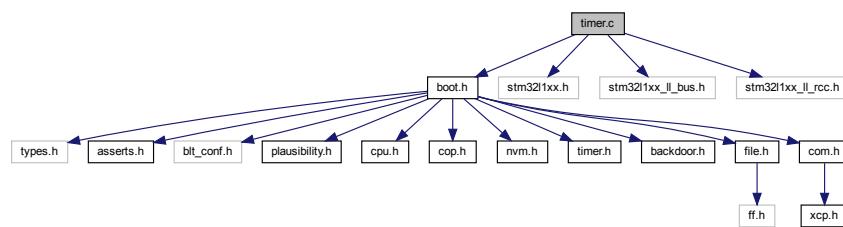
none.

Referenced by [TimerGet\(\)](#).

7.266 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32l1xx.h"
#include "stm32l1xx_ll_bus.h"
#include "stm32l1xx_ll_rcc.h"
Include dependency graph for ARMCM3_STM32L1/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into its default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- `static blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- `static blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.266.1 Detailed Description

Bootloader timer driver source file.

7.266.2 Function Documentation

7.266.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.266.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.266.2.3 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.266.2.4 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.266.2.5 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

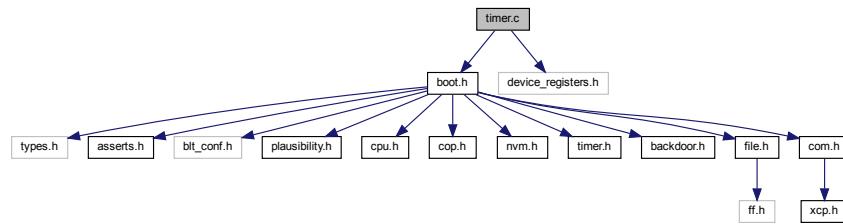
none.

Referenced by [TimerGet\(\)](#).

7.267 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "device_registers.h"
Include dependency graph for ARMCM4_S32K14/timer.c:
```



Functions

- void [TimerInit](#) (void)
Initializes the polling based millisecond timer driver.
- void [TimerReset](#) (void)
Reset the timer by placing the timer back into its default reset configuration.
- void [TimerUpdate](#) (void)
Updates the millisecond timer.
- [blt_int32u TimerGet](#) (void)
Obtains the counter value of the millisecond timer.

Variables

- [blt_int32u SystemCoreClock](#)
The system clock frequency supplied to the SysTick timer and the processor core clock.
- static [blt_int32u millisecond_counter](#)
Local variable for storing the number of milliseconds that have elapsed since startup.
- static [blt_int32u free_running_counter_last](#)
Buffer for storing the last value of the free running counter.
- static [blt_int32u counts_per_millisecond](#)
Stores the number of counts of the free running counter that equals one millisecond.

7.267.1 Detailed Description

Bootloader timer driver source file.

7.267.2 Function Documentation

7.267.2.1 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

7.267.2.2 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Ideally a 100 kHz free running counter is used as the foundation for the timer modules, as this gives 10us ticks that can be reused by other modules. The S32K14 timers unfortunately do not offer a flexible prescaler for their timers to realize such a 100 kHz free running counter. For this reason, the SysTick counter is used instead. Its 24-bit free running down-counter runs at the system speed.

Returns

none.

7.267.2.3 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

Referenced by [TimerInit\(\)](#).

7.267.2.4 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

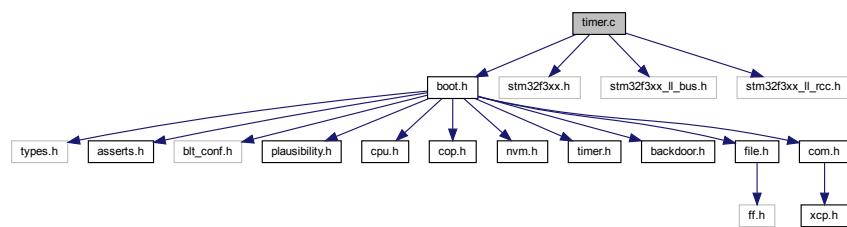
none.

Referenced by [TimerGet\(\)](#).

7.268 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32f3xx.h"
#include "stm32f3xx_ll_bus.h"
#include "stm32f3xx_ll_rcc.h"
Include dependency graph for ARMCM4_STM32F3/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into its default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- static `blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- static `blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.268.1 Detailed Description

Bootloader timer driver source file.

7.268.2 Function Documentation

7.268.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.268.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.268.2.3 TimerInit()

```
void TimerInit (
    void  )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.268.2.4 TimerReset()

```
void TimerReset (
    void  )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.268.2.5 TimerUpdate()

```
void TimerUpdate (
    void  )
```

Updates the millisecond timer.

Returns

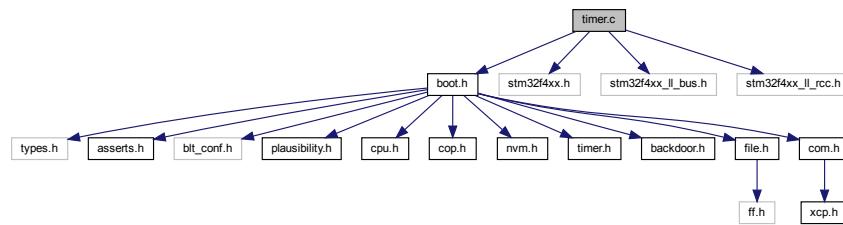
none.

Referenced by [TimerGet\(\)](#).

7.269 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32f4xx.h"
#include "stm32f4xx_ll_bus.h"
#include "stm32f4xx_ll_rcc.h"
Include dependency graph for ARMCM4_STM32F4/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into it's default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- `static blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- `static blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.269.1 Detailed Description

Bootloader timer driver source file.

7.269.2 Function Documentation

7.269.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.269.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.269.2.3 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.269.2.4 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.269.2.5 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

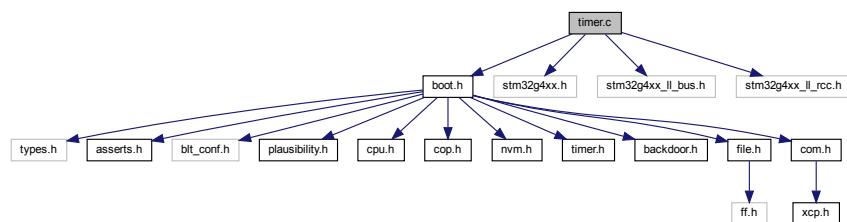
none.

Referenced by [TimerGet\(\)](#).

7.270 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32g4xx.h"
#include "stm32g4xx_ll_bus.h"
#include "stm32g4xx_ll_rcc.h"
Include dependency graph for ARMCM4_STM32G4/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into its default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- `static blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- `static blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.270.1 Detailed Description

Bootloader timer driver source file.

7.270.2 Function Documentation

7.270.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.270.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.270.2.3 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.270.2.4 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.270.2.5 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

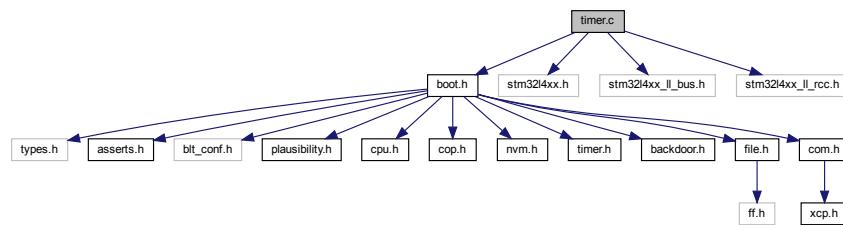
none.

Referenced by [TimerGet\(\)](#).

7.271 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32l4xx.h"
#include "stm32l4xx_ll_bus.h"
#include "stm32l4xx_ll_rcc.h"
Include dependency graph for ARMCM4_STM32L4/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into its default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- static `blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- static `blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.271.1 Detailed Description

Bootloader timer driver source file.

7.271.2 Function Documentation

7.271.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.271.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.271.2.3 TimerInit()

```
void TimerInit (
    void  )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.271.2.4 TimerReset()

```
void TimerReset (
    void  )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.271.2.5 TimerUpdate()

```
void TimerUpdate (
    void  )
```

Updates the millisecond timer.

Returns

none.

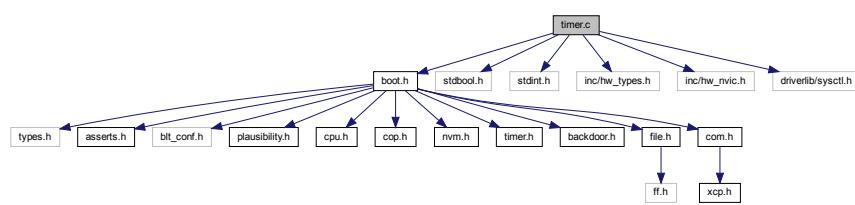
Referenced by [TimerGet\(\)](#).

7.272 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_types.h"
#include "inc/hw_nvic.h"
#include "driverlib/sysctl.h"

Include dependency graph for ARMCM4_TM4C/timer.c:
```



Functions

- void [TimerInit](#) (void)
Initializes the polling based millisecond timer driver.
- void [TimerReset](#) (void)
Reset the timer by placing the timer back into it's default reset configuration.
- void [TimerUpdate](#) (void)
Updates the millisecond timer.
- [blt_int32u TimerGet](#) (void)
Obtains the counter value of the millisecond timer.

Variables

- static [blt_int32u millisecond_counter](#)
Local variable for storing the number of milliseconds that have elapsed since startup.
- static [blt_int32u free_running_counter_last](#)
Buffer for storing the last value of the free running counter.
- static [blt_int32u counts_per_millisecond](#)
Stores the number of counts of the free running counter that equals one millisecond.

7.272.1 Detailed Description

Bootloader timer driver source file.

7.272.2 Function Documentation

7.272.2.1 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

7.272.2.2 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Returns

none.

7.272.2.3 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

Referenced by [TimerInit\(\)](#).

7.272.2.4 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

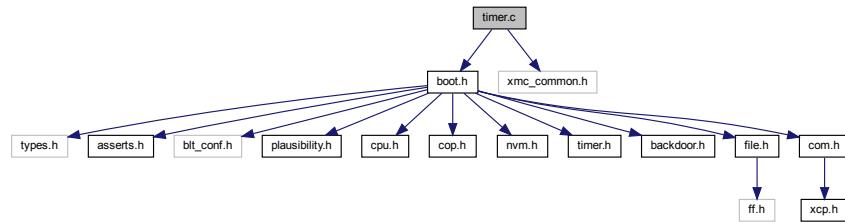
none.

Referenced by [TimerGet\(\)](#).

7.273 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "xmc_common.h"
Include dependency graph for ARMCM4_XMC4/timer.c:
```



Functions

- void [TimerInit](#) (void)
Initializes the polling based millisecond timer driver.
- void [TimerReset](#) (void)
Reset the timer by placing the timer back into it's default reset configuration.
- void [TimerUpdate](#) (void)
Updates the millisecond timer.
- [blt_int32u TimerGet](#) (void)
Obtains the counter value of the millisecond timer.

Variables

- static [blt_int32u millisecond_counter](#)
Local variable for storing the number of milliseconds that have elapsed since startup.
- static [blt_int32u free_running_counter_last](#)
Buffer for storing the last value of the free running counter.
- static [blt_int32u counts_per_millisecond](#)
Stores the number of counts of the free running counter that equals one millisecond.

7.273.1 Detailed Description

Bootloader timer driver source file.

7.273.2 Function Documentation

7.273.2.1 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

7.273.2.2 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Ideally a 100 kHz free running counter is used as the foundation for the timer modules, as this gives 10us ticks that can be reused by other modules. The XMC4 timers unfortunately do not offer a flexible prescaler for their timers to realize such a 100 kHz free running counter. For this reason, the SysTick counter is used instead. Its 24-bit free running down-counter runs at the system speed.

Returns

none.

7.273.2.3 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

Referenced by [TimerInit\(\)](#).

7.273.2.4 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

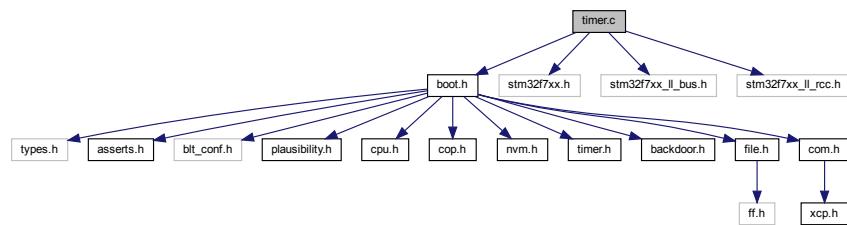
none.

Referenced by [TimerGet\(\)](#).

7.274 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32f7xx.h"
#include "stm32f7xx_ll_bus.h"
#include "stm32f7xx_ll_rcc.h"
Include dependency graph for ARMCM7_STM32F7/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into its default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- static `blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- static `blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.274.1 Detailed Description

Bootloader timer driver source file.

7.274.2 Function Documentation

7.274.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.274.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.274.2.3 TimerInit()

```
void TimerInit (
    void  )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.274.2.4 TimerReset()

```
void TimerReset (
    void  )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.274.2.5 TimerUpdate()

```
void TimerUpdate (
    void  )
```

Updates the millisecond timer.

Returns

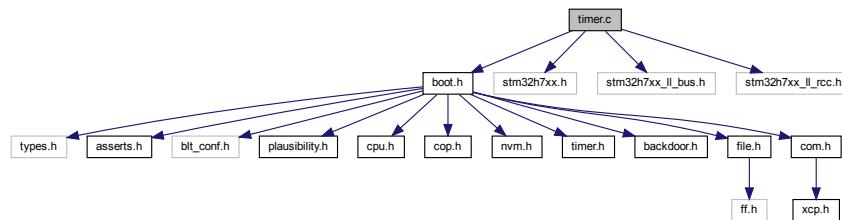
none.

Referenced by [TimerGet\(\)](#).

7.275 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "stm32h7xx.h"
#include "stm32h7xx_ll_bus.h"
#include "stm32h7xx_ll_rcc.h"
Include dependency graph for ARMCM7_STM32H7/timer.c:
```



Macros

- `#define TIMER_COUNTER_FREQ_HZ (100000U)`
Frequency of the configured timer peripheral's free running counter in Hz. This should be 100 kHz.
- `#define TIMER_COUNTS_PER_MS (TIMER_COUNTER_FREQ_HZ / 1000U)`
Number of free running counter counts that equal one millisecond.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into it's default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.
- `uint32_t HAL_GetTick (void)`
Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Variables

- `static blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- `static blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.

7.275.1 Detailed Description

Bootloader timer driver source file.

7.275.2 Function Documentation

7.275.2.1 HAL_GetTick()

```
uint32_t HAL_GetTick (
    void )
```

Override for the HAL driver's GetTick() functionality. This is needed because the bootloader doesn't use interrupts, but the HAL's tick functionality assumes that it does. This will cause the HAL_Delay() function to not work properly. As a result of this override, the HAL's tick functionality works in polling mode.

Returns

Current value of the millisecond timer.

7.275.2.2 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [HAL_GetTick\(\)](#).

7.275.2.3 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

7.275.2.4 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

7.275.2.5 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

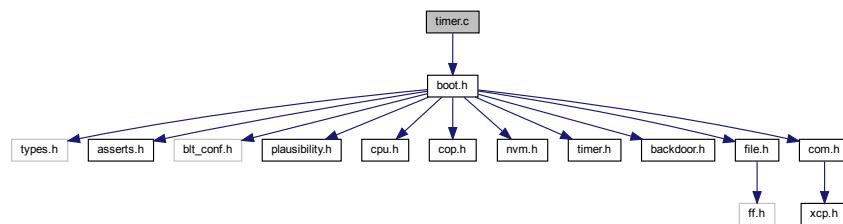
none.

Referenced by [TimerGet\(\)](#).

7.276 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
Include dependency graph for HCS12/timer.c:
```



Data Structures

- struct [tTimerRegs](#)

Structure type with the layout of the timer related control registers.

Macros

- `#define TIMER_REGS_BASE_ADDRESS (0x0040)`
Base address for the timer related control registers.
- `#define TIMER ((volatile tTimerRegs *)TIMER_REGS_BASE_ADDRESS)`
Macro for accessing the flash related control registers.
- `#define TEN_BIT (0x80)`
TSCR1 - timer enable bit.
- `#define PR0_BIT (0x01)`
TSCR2 - timer prescaler select bit 0.
- `#define PR1_BIT (0x02)`
TSCR2 - timer prescaler select bit 1.
- `#define PR2_BIT (0x04)`
TSCR2 - timer prescaler select bit 2.

Functions

- `void TimerInit (void)`
Initializes the polling based millisecond timer driver.
- `void TimerReset (void)`
Reset the timer by placing the timer back into it's default reset configuration.
- `void TimerUpdate (void)`
Updates the millisecond timer.
- `blt_int32u TimerGet (void)`
Obtains the counter value of the millisecond timer.

Variables

- `static blt_int32u millisecond_counter`
Local variable for storing the number of milliseconds that have elapsed since startup.
- `static blt_int16u free_running_counter_last`
Buffer for storing the last value of the free running counter.
- `static blt_int16u counts_per_millisecond`
Stores the number of counts of the free running counter that equals one millisecond.

7.276.1 Detailed Description

Bootloader timer driver source file.

7.276.2 Function Documentation

7.276.2.1 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

7.276.2.2 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Returns

none.

7.276.2.3 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

Referenced by [TimerInit\(\)](#).

7.276.2.4 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

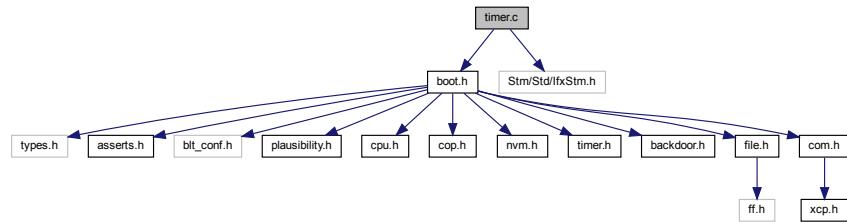
none.

Referenced by [TimerGet\(\)](#).

7.277 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "Stm/Std/IfxStm.h"
Include dependency graph for TRICORE_TC2/timer.c:
```



Functions

- void [TimerInit](#) (void)
Initializes the polling based millisecond timer driver.
- void [TimerReset](#) (void)
Reset the timer by placing the timer back into it's default reset configuration.
- void [TimerUpdate](#) (void)
Updates the millisecond timer.
- [blt_int32u TimerGet](#) (void)
Obtains the counter value of the millisecond timer.

Variables

- static [blt_int32u millisecond_counter](#)
Local variable for storing the number of milliseconds that have elapsed since startup.
- static [blt_int32u free_running_counter_last](#)
Buffer for storing the last value of the lower 32-bits of the free running counter.
- static [blt_int32u counts_per_millisecond](#)
Stores the number of counts of the free running counter that equals one millisecond.

7.277.1 Detailed Description

Bootloader timer driver source file.

7.277.2 Function Documentation

7.277.2.1 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

7.277.2.2 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Returns

none.

7.277.2.3 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

Referenced by [TimerInit\(\)](#).

7.277.2.4 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

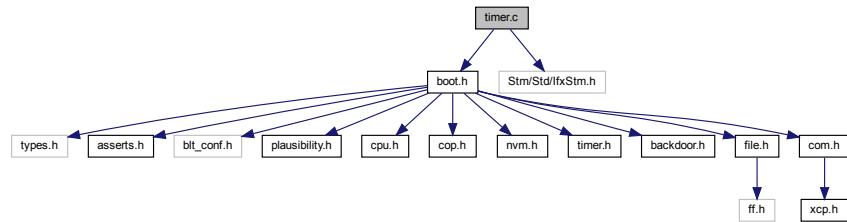
none.

Referenced by [TimerGet\(\)](#).

7.278 timer.c File Reference

Bootloader timer driver source file.

```
#include "boot.h"
#include "Stm/Std/IfxStm.h"
Include dependency graph for TRICORE_TC3/timer.c:
```



Functions

- void [TimerInit](#) (void)
Initializes the polling based millisecond timer driver.
- void [TimerReset](#) (void)
Reset the timer by placing the timer back into it's default reset configuration.
- void [TimerUpdate](#) (void)
Updates the millisecond timer.
- [blt_int32u TimerGet](#) (void)
Obtains the counter value of the millisecond timer.

Variables

- static [blt_int32u millisecond_counter](#)
Local variable for storing the number of milliseconds that have elapsed since startup.
- static [blt_int32u free_running_counter_last](#)
Buffer for storing the last value of the lower 32-bits of the free running counter.
- static [blt_int32u counts_per_millisecond](#)
Stores the number of counts of the free running counter that equals one millisecond.

7.278.1 Detailed Description

Bootloader timer driver source file.

7.278.2 Function Documentation

7.278.2.1 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

7.278.2.2 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Returns

none.

7.278.2.3 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

Referenced by [TimerInit\(\)](#).

7.278.2.4 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

none.

Referenced by [TimerGet\(\)](#).

7.279 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- typedef unsigned char **blt_bool**
- typedef char **blt_char**
- typedef unsigned long **blt_addr**
- typedef unsigned char **blt_int8u**
- typedef signed char **blt_int8s**
- typedef unsigned short **blt_int16u**
- typedef signed short **blt_int16s**
- typedef unsigned int **blt_int32u**
- typedef signed int **blt_int32s**

7.279.1 Detailed Description

Bootloader types header file.

7.279.2 Typedef Documentation

7.279.2.1 blt_addr

```
typedef unsigned long blt_addr  
memory address type
```

7.279.2.2 blt_bool

```
typedef unsigned char blt_bool  
boolean type
```

7.279.2.3 blt_char

```
typedef char blt_char
```

character type

7.279.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.279.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.279.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.279.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.279.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.279.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.280 types.h File Reference

Bootloader types header file.

Macros

- `#define BLT_TRUE (1)`
Boolean true value.
- `#define BLT_FALSE (0)`
Boolean false value.
- `#define BLT_NULL ((void *)0)`
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`

7.280.1 Detailed Description

Bootloader types header file.

7.280.2 Typedef Documentation

7.280.2.1 blt_addr

```
typedef unsigned long blt_addr
```

memory address type

7.280.2.2 blt_bool

```
typedef unsigned char blt_bool
```

boolean type

7.280.2.3 blt_char

```
typedef char blt_char
```

character type

7.280.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.280.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.280.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.280.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.280.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.280.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.281 types.h File Reference

Bootloader types header file.

Macros

- `#define BLT_TRUE (1)`
Boolean true value.
- `#define BLT_FALSE (0)`
Boolean false value.
- `#define BLT_NULL ((void *)0)`
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`
- `typedef unsigned long long blt_int64u`
- `typedef signed long long blt_int64s`

7.281.1 Detailed Description

Bootloader types header file.

7.281.2 Typedef Documentation

7.281.2.1 blt_addr

```
typedef unsigned long blt_addr
```

memory address type

7.281.2.2 blt_bool

```
typedef unsigned char blt_bool
```

boolean type

7.281.2.3 blt_char

```
typedef char blt_char
```

character type

7.281.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.281.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.281.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.281.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.281.2.8 blt_int64s

```
typedef signed long long blt_int64s
```

64-bit signed integer

7.281.2.9 blt_int64u

```
typedef unsigned long long blt_int64u
```

64-bit unsigned integer

7.281.2.10 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.281.2.11 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.282 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- typedef unsigned char **blt_bool**
- typedef char **blt_char**
- typedef unsigned long **blt_addr**
- typedef unsigned char **blt_int8u**
- typedef signed char **blt_int8s**
- typedef unsigned short **blt_int16u**
- typedef signed short **blt_int16s**
- typedef unsigned int **blt_int32u**
- typedef signed int **blt_int32s**

7.282.1 Detailed Description

Bootloader types header file.

7.282.2 Typedef Documentation

7.282.2.1 blt_addr

```
typedef unsigned long blt_addr
```

memory address type

7.282.2.2 blt_bool

```
typedef unsigned char blt_bool
```

boolean type

7.282.2.3 blt_char

```
typedef char blt_char
```

character type

7.282.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.282.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.282.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.282.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.282.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.282.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.283 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`
- `typedef unsigned long long blt_int64u`
- `typedef signed long long blt_int64s`

7.283.1 Detailed Description

Bootloader types header file.

7.283.2 Typedef Documentation

7.283.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.283.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.283.2.3 blt_char

`typedef char blt_char`

character type

7.283.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.283.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.283.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.283.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.283.2.8 blt_int64s

```
typedef signed long long blt_int64s
```

64-bit signed integer

7.283.2.9 blt_int64u

```
typedef unsigned long long blt_int64u
```

64-bit unsigned integer

7.283.2.10 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.283.2.11 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.284 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- typedef unsigned char **blt_bool**
- typedef char **blt_char**
- typedef unsigned long **blt_addr**
- typedef unsigned char **blt_int8u**
- typedef signed char **blt_int8s**
- typedef unsigned short **blt_int16u**
- typedef signed short **blt_int16s**
- typedef unsigned int **blt_int32u**
- typedef signed int **blt_int32s**

7.284.1 Detailed Description

Bootloader types header file.

7.284.2 Typedef Documentation

7.284.2.1 blt_addr

```
typedef unsigned long blt_addr
```

memory address type

7.284.2.2 blt_bool

```
typedef unsigned char blt_bool
```

boolean type

7.284.2.3 blt_char

```
typedef char blt_char
```

character type

7.284.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.284.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.284.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.284.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.284.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.284.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.285 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- typedef unsigned char **blt_bool**
- typedef char **blt_char**
- typedef unsigned long **blt_addr**
- typedef unsigned char **blt_int8u**
- typedef signed char **blt_int8s**
- typedef unsigned short **blt_int16u**
- typedef signed short **blt_int16s**
- typedef unsigned int **blt_int32u**
- typedef signed int **blt_int32s**

7.285.1 Detailed Description

Bootloader types header file.

7.285.2 Typedef Documentation

7.285.2.1 blt_addr

```
typedef unsigned long blt_addr
```

memory address type

7.285.2.2 blt_bool

```
typedef unsigned char blt_bool
```

boolean type

7.285.2.3 blt_char

```
typedef char blt_char
```

character type

7.285.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.285.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.285.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.285.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.285.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.285.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.286 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

TypeDefs

- typedef unsigned char blt_bool
- typedef char blt_char
- typedef unsigned long blt_addr
- typedef unsigned char blt_int8u
- typedef signed char blt_int8s
- typedef unsigned short blt_int16u
- typedef signed short blt_int16s
- typedef unsigned int blt_int32u
- typedef signed int blt_int32s
- typedef unsigned long long blt_int64u
- typedef signed long long blt_int64s

7.286.1 Detailed Description

Bootloader types header file.

7.286.2 Typedef Documentation

7.286.2.1 blt_addr

```
typedef unsigned long blt_addr
```

memory address type

7.286.2.2 blt_bool

```
typedef unsigned char blt_bool
```

boolean type

7.286.2.3 blt_char

```
typedef char blt_char
```

character type

7.286.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.286.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.286.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.286.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.286.2.8 blt_int64s

```
typedef signed long long blt_int64s
```

64-bit signed integer

7.286.2.9 blt_int64u

```
typedef unsigned long long blt_int64u
```

64-bit unsigned integer

7.286.2.10 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.286.2.11 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.287 types.h File Reference

Bootloader types header file.

Macros

- `#define BLT_TRUE (1)`
Boolean true value.
- `#define BLT_FALSE (0)`
Boolean false value.
- `#define BLT_NULL ((void *)0)`
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`
- `typedef unsigned long long blt_int64u`
- `typedef signed long long blt_int64s`

7.287.1 Detailed Description

Bootloader types header file.

7.287.2 Typedef Documentation

7.287.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.287.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.287.2.3 blt_char

```
typedef char blt_char
```

character type

7.287.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.287.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.287.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.287.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.287.2.8 blt_int64s

```
typedef signed long long blt_int64s
```

64-bit signed integer

7.287.2.9 blt_int64u

```
typedef unsigned long long blt_int64u
```

64-bit unsigned integer

7.287.2.10 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.287.2.11 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.288 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- typedef unsigned char **blt_bool**
- typedef char **blt_char**
- typedef unsigned long **blt_addr**
- typedef unsigned char **blt_int8u**
- typedef signed char **blt_int8s**
- typedef unsigned short **blt_int16u**
- typedef signed short **blt_int16s**
- typedef unsigned int **blt_int32u**
- typedef signed int **blt_int32s**
- typedef unsigned long long **blt_int64u**
- typedef signed long long **blt_int64s**

7.288.1 Detailed Description

Bootloader types header file.

7.288.2 Typedef Documentation

7.288.2.1 blt_addr

```
typedef unsigned long blt_addr
```

memory address type

7.288.2.2 blt_bool

```
typedef unsigned char blt_bool
```

boolean type

7.288.2.3 blt_char

```
typedef char blt_char
```

character type

7.288.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.288.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.288.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.288.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.288.2.8 blt_int64s

```
typedef signed long long blt_int64s
```

64-bit signed integer

7.288.2.9 blt_int64u

```
typedef unsigned long long blt_int64u
```

64-bit unsigned integer

7.288.2.10 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.288.2.11 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.289 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`

7.289.1 Detailed Description

Bootloader types header file.

7.289.2 Typedef Documentation

7.289.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.289.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.289.2.3 blt_char

`typedef char blt_char`

character type

7.289.2.4 blt_int16s

`typedef signed short blt_int16s`

16-bit signed integer

7.289.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.289.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.289.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.289.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.289.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.290 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`

7.290.1 Detailed Description

Bootloader types header file.

7.290.2 Typedef Documentation

7.290.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.290.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.290.2.3 blt_char

`typedef char blt_char`

character type

7.290.2.4 blt_int16s

`typedef signed short blt_int16s`

16-bit signed integer

7.290.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.290.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.290.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.290.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.290.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.291 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`

7.291.1 Detailed Description

Bootloader types header file.

7.291.2 Typedef Documentation

7.291.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.291.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.291.2.3 blt_char

`typedef char blt_char`

character type

7.291.2.4 blt_int16s

`typedef signed short blt_int16s`

16-bit signed integer

7.291.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.291.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.291.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.291.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.291.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.292 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`

7.292.1 Detailed Description

Bootloader types header file.

7.292.2 Typedef Documentation

7.292.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.292.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.292.2.3 blt_char

`typedef char blt_char`

character type

7.292.2.4 blt_int16s

`typedef signed short blt_int16s`

16-bit signed integer

7.292.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.292.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.292.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.292.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.292.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.293 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`

7.293.1 Detailed Description

Bootloader types header file.

7.293.2 Typedef Documentation

7.293.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.293.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.293.2.3 blt_char

`typedef char blt_char`

character type

7.293.2.4 blt_int16s

`typedef signed short blt_int16s`

16-bit signed integer

7.293.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.293.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.293.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.293.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.293.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.294 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`

7.294.1 Detailed Description

Bootloader types header file.

7.294.2 Typedef Documentation

7.294.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.294.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.294.2.3 blt_char

`typedef char blt_char`

character type

7.294.2.4 blt_int16s

`typedef signed short blt_int16s`

16-bit signed integer

7.294.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.294.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.294.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.294.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.294.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.295 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`

7.295.1 Detailed Description

Bootloader types header file.

7.295.2 Typedef Documentation

7.295.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.295.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.295.2.3 blt_char

`typedef char blt_char`

character type

7.295.2.4 blt_int16s

`typedef signed short blt_int16s`

16-bit signed integer

7.295.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.295.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.295.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.295.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.295.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.296 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`

7.296.1 Detailed Description

Bootloader types header file.

7.296.2 Typedef Documentation

7.296.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.296.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.296.2.3 blt_char

`typedef char blt_char`

character type

7.296.2.4 blt_int16s

`typedef signed short blt_int16s`

16-bit signed integer

7.296.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.296.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.296.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.296.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.296.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.297 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`
- `typedef unsigned long long blt_int64u`
- `typedef signed long long blt_int64s`

7.297.1 Detailed Description

Bootloader types header file.

7.297.2 Typedef Documentation

7.297.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.297.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.297.2.3 blt_char

`typedef char blt_char`

character type

7.297.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.297.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.297.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.297.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.297.2.8 blt_int64s

```
typedef signed long long blt_int64s
```

64-bit signed integer

7.297.2.9 blt_int64u

```
typedef unsigned long long blt_int64u
```

64-bit unsigned integer

7.297.2.10 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.297.2.11 blt_int8u

typedef unsigned char `blt_int8u`

8-bit unsigned integer

7.298 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- typedef unsigned char `blt_bool`
- typedef char `blt_char`
- typedef unsigned long `blt_addr`
- typedef unsigned char `blt_int8u`
- typedef signed char `blt_int8s`
- typedef unsigned short `blt_int16u`
- typedef signed short `blt_int16s`
- typedef unsigned int `blt_int32u`
- typedef signed int `blt_int32s`

7.298.1 Detailed Description

Bootloader types header file.

7.298.2 Typedef Documentation

7.298.2.1 blt_addr

```
typedef unsigned long blt_addr
```

memory address type

7.298.2.2 blt_bool

```
typedef unsigned char blt_bool
```

boolean type

7.298.2.3 blt_char

```
typedef char blt_char
```

character type

7.298.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.298.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.298.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.298.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.298.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.298.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.299 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- typedef unsigned char **blt_bool**
- typedef char **blt_char**
- typedef unsigned long **blt_addr**
- typedef unsigned char **blt_int8u**
- typedef signed char **blt_int8s**
- typedef unsigned short **blt_int16u**
- typedef signed short **blt_int16s**
- typedef unsigned int **blt_int32u**
- typedef signed int **blt_int32s**

7.299.1 Detailed Description

Bootloader types header file.

7.299.2 Typedef Documentation

7.299.2.1 blt_addr

```
typedef unsigned long blt_addr
```

memory address type

7.299.2.2 blt_bool

```
typedef unsigned char blt_bool
```

boolean type

7.299.2.3 blt_char

```
typedef char blt_char
```

character type

7.299.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.299.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.299.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.299.2.7 blt_int32u

typedef unsigned int `blt_int32u`

32-bit unsigned integer

7.299.2.8 blt_int8s

typedef signed char `blt_int8s`

8-bit signed integer

7.299.2.9 blt_int8u

typedef unsigned char `blt_int8u`

8-bit unsigned integer

7.300 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- typedef unsigned char `blt_bool`
- typedef char `blt_char`
- typedef unsigned long `blt_addr`
- typedef unsigned char `blt_int8u`
- typedef signed char `blt_int8s`
- typedef unsigned short `blt_int16u`
- typedef signed short `blt_int16s`
- typedef unsigned int `blt_int32u`
- typedef signed int `blt_int32s`

7.300.1 Detailed Description

Bootloader types header file.

7.300.2 Typedef Documentation

7.300.2.1 blt_addr

```
typedef unsigned long blt_addr
```

memory address type

7.300.2.2 blt_bool

```
typedef unsigned char blt_bool
```

boolean type

7.300.2.3 blt_char

```
typedef char blt_char
```

character type

7.300.2.4 blt_int16s

```
typedef signed short blt_int16s
```

16-bit signed integer

7.300.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.300.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.300.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.300.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.300.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.301 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`

7.301.1 Detailed Description

Bootloader types header file.

7.301.2 Typedef Documentation

7.301.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.301.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.301.2.3 blt_char

`typedef char blt_char`

character type

7.301.2.4 blt_int16s

`typedef signed short blt_int16s`

16-bit signed integer

7.301.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.301.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.301.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.301.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.301.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.302 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`

7.302.1 Detailed Description

Bootloader types header file.

7.302.2 Typedef Documentation

7.302.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.302.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.302.2.3 blt_char

`typedef char blt_char`

character type

7.302.2.4 blt_int16s

`typedef signed short blt_int16s`

16-bit signed integer

7.302.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.302.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.302.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.302.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.302.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.303 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned long blt_int32u`
- `typedef signed long blt_int32s`

7.303.1 Detailed Description

Bootloader types header file.

7.303.2 Typedef Documentation

7.303.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.303.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.303.2.3 blt_char

`typedef char blt_char`

character type

7.303.2.4 blt_int16s

`typedef signed short blt_int16s`

16-bit signed integer

7.303.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.303.2.6 blt_int32s

```
typedef signed long blt_int32s
```

32-bit signed integer

7.303.2.7 blt_int32u

```
typedef unsigned long blt_int32u
```

32-bit unsigned integer

7.303.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.303.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.304 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`

7.304.1 Detailed Description

Bootloader types header file.

7.304.2 Typedef Documentation

7.304.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.304.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.304.2.3 blt_char

`typedef char blt_char`

character type

7.304.2.4 blt_int16s

`typedef signed short blt_int16s`

16-bit signed integer

7.304.2.5 blt_int16u

```
typedef unsigned short blt_int16u
```

16-bit unsigned integer

7.304.2.6 blt_int32s

```
typedef signed int blt_int32s
```

32-bit signed integer

7.304.2.7 blt_int32u

```
typedef unsigned int blt_int32u
```

32-bit unsigned integer

7.304.2.8 blt_int8s

```
typedef signed char blt_int8s
```

8-bit signed integer

7.304.2.9 blt_int8u

```
typedef unsigned char blt_int8u
```

8-bit unsigned integer

7.305 types.h File Reference

Bootloader types header file.

Macros

- #define **BLT_TRUE** (1)
Boolean true value.
- #define **BLT_FALSE** (0)
Boolean false value.
- #define **BLT_NULL** ((void *)0)
NULL pointer value.

Typedefs

- `typedef unsigned char blt_bool`
- `typedef char blt_char`
- `typedef unsigned long blt_addr`
- `typedef unsigned char blt_int8u`
- `typedef signed char blt_int8s`
- `typedef unsigned short blt_int16u`
- `typedef signed short blt_int16s`
- `typedef unsigned int blt_int32u`
- `typedef signed int blt_int32s`

7.305.1 Detailed Description

Bootloader types header file.

7.305.2 Typedef Documentation

7.305.2.1 blt_addr

`typedef unsigned long blt_addr`

memory address type

7.305.2.2 blt_bool

`typedef unsigned char blt_bool`

boolean type

7.305.2.3 blt_char

`typedef char blt_char`

character type

7.305.2.4 blt_int16s

`typedef signed short blt_int16s`

16-bit signed integer

7.305.2.5 blt_int16u

`typedef unsigned short blt_int16u`

16-bit unsigned integer

7.305.2.6 blt_int32s

`typedef signed int blt_int32s`

32-bit signed integer

7.305.2.7 blt_int32u

`typedef unsigned int blt_int32u`

32-bit unsigned integer

7.305.2.8 blt_int8s

`typedef signed char blt_int8s`

8-bit signed integer

7.305.2.9 blt_int8u

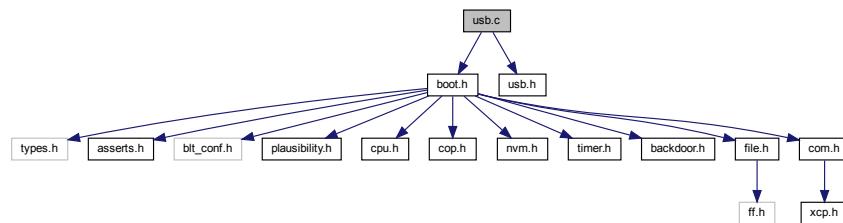
`typedef unsigned char blt_int8u`

8-bit unsigned integer

7.306 usb.c File Reference

Bootloader USB communication interface source file.

```
#include "boot.h"
#include "usb.h"
Include dependency graph for _template/usb.c:
```



Data Structures

- struct **tFifoCtrl**
Structure type for fifo control.
- struct **tFifoPipe**
Structure type for a fifo pipe.

Macros

- #define **FIFO_MAX_BUFFERS** (2)
Total number of fifo buffers.
- #define **FIFO_ERR_INVALID_HANDLE** (255)
Invalid value for a fifo buffer handle.
- #define **FIFO_PIPE_SIZE** (64)
Number of bytes that fit in the fifo pipe.
- #define **BULK_DATA_MAX_PACKET_SIZE** (64)
Endpoint IN & OUT Packet size.

Functions

- static **blt_bool UsbReceiveByte** (**blt_int8u** *data)
Receives a communication interface byte if one is present.
- static **blt_bool UsbTransmitByte** (**blt_int8u** data)
Transmits a communication interface byte.
- static void **UsbFifoMgrInit** (void)
Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.
- static **blt_int8u UsbFifoMgrCreate** (**blt_int8u** *buffer, **blt_int8u** length)
Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.
- static **blt_bool UsbFifoMgrWrite** (**blt_int8u** handle, **blt_int8u** data)
Stores data in the fifo.
- static **blt_bool UsbFifoMgrRead** (**blt_int8u** handle, **blt_int8u** *data)
Retrieves data from the fifo.
- static **blt_int8u UsbFifoMgrScan** (**blt_int8u** handle)
Returns the number of data entries currently present in the fifo.
- void **UsbInit** (void)
Initializes the USB communication interface.
- void **UsbFree** (void)
Releases the USB communication interface.
- void **UsbTransmitPacket** (**blt_int8u** *data, **blt_int8u** len)
Transmits a packet formatted for the communication interface.
- **blt_bool UsbReceivePacket** (**blt_int8u** *data, **blt_int8u** *len)
Receives a communication interface packet if one is present.
- void **UsbTransmitPipeBulkIN** (void)
Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.
- void **UsbReceivePipeBulkOUT** (void)
Stores data that was received on the Bulk OUT pipe in the fifo.

Variables

- static tFifoCtrl **fifoCtrl** [FIFO_MAX_BUFFERS]
Local variable that holds the fifo control structures.
- static tFifoCtrl * **fifoCtrlFree**
Local pointer that points to the next free fifo control structure.
- static tFifoPipe **fifoPipeBulkIN**
Fifo pipe used for the bulk in endpoint.
- static tFifoPipe **fifoPipeBulkOUT**
Fifo pipe used for the bulk out endpoint.

7.306.1 Detailed Description

Bootloader USB communication interface source file.

7.306.2 Function Documentation

7.306.2.1 UsbFifoMgrCreate()

```
static blt_int8u UsbFifoMgrCreate (
    blt_int8u * buffer,
    blt_int8u length ) [static]
```

Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.

Parameters

| | |
|---------------|---|
| <i>buffer</i> | Pointer to the first element in the data storage fifo. |
| <i>length</i> | Maximum number of data elements that can be stored in the fifo. |

Returns

Fifo handle if successfull, or FIFO_ERR_INVALID_HANDLE.

Referenced by [UsbInit\(\)](#).

7.306.2.2 UsbFifoMgrInit()

```
static void UsbFifoMgrInit (
    void ) [static]
```

Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into `fifoCtrl[]`. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in `fifoCtrl[]` this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.

Returns

none.

Referenced by [UsbInit\(\)](#).

7.306.2.3 UsbFifoMgrRead()

```
static blt_bool UsbFifoMgrRead (
    blt_int8u handle,
    blt_int8u * data ) [static]
```

Retrieves data from the fifo.

Parameters

| | |
|---------------------|---|
| <code>handle</code> | Identifies the fifo to read data from. |
| <code>data</code> | Pointer to where the read data is to be stored. |

Returns

`BLT_TRUE` if the data was successfully read from the fifo, `BLT_FALSE` otherwise.

Referenced by [UsbReceiveByte\(\)](#), and [UsbTransmitPipeBulkIN\(\)](#).

7.306.2.4 UsbFifoMgrScan()

```
static blt_int8u UsbFifoMgrScan (
    blt_int8u handle ) [static]
```

Returns the number of data entries currently present in the fifo.

Parameters

| | |
|---------------------|--|
| <code>handle</code> | Identifies the fifo that is to be scanned. |
|---------------------|--|

Returns

Number of data entries in the fifo if successful, otherwise 0.

Referenced by [UsbTransmitPipeBulkIN\(\)](#).

7.306.2.5 UsbFifoMgrWrite()

```
static blt_bool UsbFifoMgrWrite (
    blt_int8u handle,
    blt_int8u data ) [static]
```

Stores data in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo to write data to. |
| <i>data</i> | Pointer to the data that is to be written to the fifo. |

Returns

BLT_TRUE if the data was successfully stored in the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceivePipeBulkOUT\(\)](#), and [UsbTransmitByte\(\)](#).

7.306.2.6 UsbFree()

```
void UsbFree (
    void )
```

Releases the USB communication interface.

Returns

none.

Referenced by [ComFree\(\)](#).

7.306.2.7 UsbInit()

```
void UsbInit (
    void )
```

Initializes the USB communication interface.

Returns

none.

Referenced by [ComInit\(\)](#).

7.306.2.8 UsbReceiveByte()

```
static blt_bool UsbReceiveByte (
    blt_int8u * data ) [static]
```

Receives a communication interface byte if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte where the data is to be stored. |
|-------------|---|

Returns

BLT_TRUE if a byte was received, BLT_FALSE otherwise.

7.306.2.9 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

Referenced by [ComTask\(\)](#).

7.306.2.10 UsbReceivePipeBulkOUT()

```
void UsbReceivePipeBulkOUT (
    void )
```

Stores data that was received on the Bulk OUT pipe in the fifo.

Returns

none.

7.306.2.11 UsbTransmitByte()

```
static blt_bool UsbTransmitByte (
    blt_int8u data ) [static]
```

Transmits a communication interface byte.

Parameters

| | |
|-------------|--|
| <i>data</i> | Value of byte that is to be transmitted. |
|-------------|--|

Returns

BLT_TRUE if the byte was transmitted, BLT_FALSE otherwise.

Referenced by [UsbTransmitPacket\(\)](#).

7.306.2.12 UsbTransmitPacket()

```
void UsbTransmitPacket ( 
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

Referenced by [ComTransmitPacket\(\)](#).

7.306.2.13 UsbTransmitPipeBulkIN()

```
void UsbTransmitPipeBulkIN ( 
    void )
```

Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.

Returns

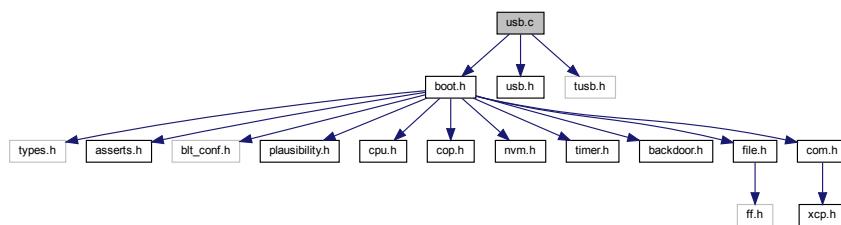
none.

7.307 usb.c File Reference

Bootloader USB communication interface source file.

```
#include "boot.h"
#include "usb.h"
#include "tusb.h"
```

Include dependency graph for ARMCM33_STM32H5/usb.c:



Functions

- static `blt_bool UsbReceiveByte (blt_int8u *data)`
Receives a communication interface byte if one is present.
- void `UsbInit (void)`
Initializes the USB communication interface.
- void `UsbFree (void)`
Releases the USB communication interface.
- void `UsbTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool UsbReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.
- void `tud_mount_cb (void)`
TinyUSB stack callback invoked then the USB device is mounted.
- void `tud_umount_cb (void)`
TinyUSB stack callback invoked then the USB device is unmounted.
- void `tud_suspend_cb (bool remote_wakeup_en)`
TinyUSB stack callback invoked then the USB bus is suspended. Within 7ms the device must draw an average of current less than 2.5 mA from the bus.
- void `tud_resume_cb (void)`
TinyUSB stack callback invoked then the USB bus is resumed.

7.307.1 Detailed Description

Bootloader USB communication interface source file.

7.307.2 Function Documentation

7.307.2.1 tud_suspend_cb()

```
void tud_suspend_cb (
    bool remote_wakeup_en )
```

TinyUSB stack callback invoked then the USB bus is suspended. Within 7ms the device must draw an average of current less than 2.5 mA from the bus.

Parameters

| | |
|-------------------------|--|
| <i>remote_wakeup_en</i> | True if the host allows us to perform a remote wakeup. |
|-------------------------|--|

7.307.2.2 UsbFree()

```
void UsbFree (
    void )
```

Releases the USB communication interface.

Returns

none.

7.307.2.3 UsbInit()

```
void UsbInit (
    void )
```

Initializes the USB communication interface.

Returns

none.

7.307.2.4 UsbReceiveByte()

```
static blt_bool UsbReceiveByte (
    blt_int8u * data ) [static]
```

Receives a communication interface byte if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte where the data is to be stored. |
|-------------|---|

Returns

BLT_TRUE if a byte was received, BLT_FALSE otherwise.

7.307.2.5 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.307.2.6 UsbTransmitPacket()

```
void UsbTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

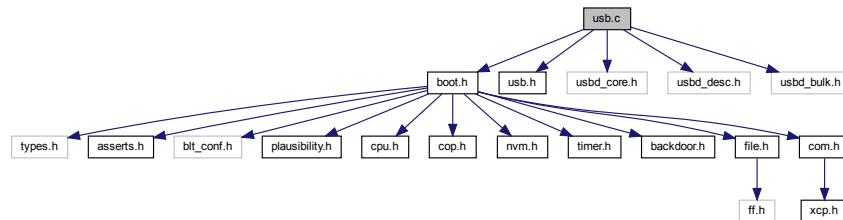
Returns

none.

7.308 usb.c File Reference

Bootloader USB communication interface source file.

```
#include "boot.h"
#include "usb.h"
#include "usbd_core.h"
#include "usbd_desc.h"
#include "usbd_bulk.h"
Include dependency graph for ARMCM33_STM32L5/usb.c:
```



Data Structures

- struct **tFifoCtrl**
Structure type for fifo control.
- struct **tFifoPipe**
Structure type for a fifo pipe.

Macros

- #define **FIFO_MAX_BUFFERS** (2)
Total number of fifo buffers.
- #define **FIFO_ERR_INVALID_HANDLE** (255)
Invalid value for a fifo buffer handle.
- #define **FIFO_PIPE_SIZE** (64)
Number of bytes that fit in the fifo pipe.

Functions

- static `blt_bool UsbReceiveByte (blt_int8u *data)`
Receives a communication interface byte if one is present.
- static `blt_bool UsbTransmitByte (blt_int8u data)`
Transmits a communication interface byte.
- static void `UsbFifoMgrInit (void)`
Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into `fifoCtrl[]`. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in `fifoCtrl[]` this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.
- static `blt_int8u UsbFifoMgrCreate (blt_int8u *buffer, blt_int8u length)`
Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.
- static `blt_bool UsbFifoMgrWrite (blt_int8u handle, blt_int8u data)`
Stores data in the fifo.
- static `blt_bool UsbFifoMgrRead (blt_int8u handle, blt_int8u *data)`
Retrieves data from the fifo.
- static `blt_int8u UsbFifoMgrScan (blt_int8u handle)`
Returns the number of data entries currently present in the fifo.
- void `UsbInit (void)`
Initializes the USB communication interface.
- void `UsbFree (void)`
Releases the USB communication interface.
- void `UsbTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool UsbReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.
- void `UsbTransmitPipeBulkIN (void)`
Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.
- void `UsbReceivePipeBulkOUT (blt_int8u epnum)`
Stores data that was received on the Bulk OUT pipe in the fifo.

Variables

- static `tFifoCtrl fifoCtrl [FIFO_MAX_BUFFERS]`
Local variable that holds the fifo control structures.
- static `tFifoCtrl * fifoCtrlFree`
Local pointer that points to the next free fifo control structure.
- static `tFifoPipe fifoPipeBulkIN`
Fifo pipe used for the bulk in endpoint.
- static `tFifoPipe fifoPipeBulkOUT`
Fifo pipe used for the bulk out endpoint.
- static `USBD_HandleTypeDef hUsbDeviceFS`
USB device handle.

7.308.1 Detailed Description

Bootloader USB communication interface source file.

7.308.2 Function Documentation

7.308.2.1 UsbFifoMgrCreate()

```
static blt_int8u UsbFifoMgrCreate (
    blt_int8u * buffer,
    blt_int8u length ) [static]
```

Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.

Parameters

| | |
|---------------|---|
| <i>buffer</i> | Pointer to the first element in the data storage fifo. |
| <i>length</i> | Maximum number of data elements that can be stored in the fifo. |

Returns

Fifo handle if successfull, or FIFO_ERR_INVALID_HANDLE.

Referenced by [UsbInit\(\)](#).

7.308.2.2 UsbFifoMgrInit()

```
static void UsbFifoMgrInit (
    void ) [static]
```

Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.

Returns

none.

Referenced by [UsbInit\(\)](#).

7.308.2.3 UsbFifoMgrRead()

```
static blt_bool UsbFifoMgrRead (
    blt_int8u handle,
    blt_int8u * data ) [static]
```

Retrieves data from the fifo.

Parameters

| | |
|---------------|---|
| <i>handle</i> | Identifies the fifo to read data from. |
| <i>data</i> | Pointer to where the read data is to be stored. |

Returns

BLT_TRUE if the data was successfully read from the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceiveByte\(\)](#), and [UsbTransmitPipeBulkIN\(\)](#).

7.308.2.4 UsbFifoMgrScan()

```
static blt_int8u UsbFifoMgrScan (
    blt_int8u handle ) [static]
```

Returns the number of data entries currently present in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo that is to be scanned. |
|---------------|--|

Returns

Number of data entries in the fifo if successful, otherwise 0.

Referenced by [UsbTransmitPipeBulkIN\(\)](#).

7.308.2.5 UsbFifoMgrWrite()

```
static blt_bool UsbFifoMgrWrite (
    blt_int8u handle,
    blt_int8u data ) [static]
```

Stores data in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo to write data to. |
| <i>data</i> | Pointer to the data that is to be written to the fifo. |

Returns

BLT_TRUE if the data was successfully stored in the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceivePipeBulkOUT\(\)](#), and [UsbTransmitByte\(\)](#).

7.308.2.6 UsbFree()

```
void UsbFree (
    void )
```

Releases the USB communication interface.

Returns

none.

7.308.2.7 UsbInit()

```
void UsbInit (
    void )
```

Initializes the USB communication interface.

Returns

none.

7.308.2.8 UsbReceiveByte()

```
static blt_bool UsbReceiveByte (
    blt_int8u * data ) [static]
```

Receives a communication interface byte if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte where the data is to be stored. |
|-------------|---|

Returns

BLT_TRUE if a byte was received, BLT_FALSE otherwise.

7.308.2.9 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.308.2.10 UsbReceivePipeBulkOUT()

```
void UsbReceivePipeBulkOUT (
    blt_int8u epnum )
```

Stores data that was received on the Bulk OUT pipe in the fifo.

Returns

none.

7.308.2.11 UsbTransmitByte()

```
static blt_bool UsbTransmitByte (
    blt_int8u data ) [static]
```

Transmits a communication interface byte.

Parameters

| | |
|-------------|--|
| <i>data</i> | Value of byte that is to be transmitted. |
|-------------|--|

Returns

BLT_TRUE if the byte was transmitted, BLT_FALSE otherwise.

Referenced by [UsbTransmitPacket\(\)](#).

7.308.2.12 UsbTransmitPacket()

```
void UsbTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.308.2.13 UsbTransmitPipeBulkIN()

```
void UsbTransmitPipeBulkIN (
    void )
```

Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.

Returns

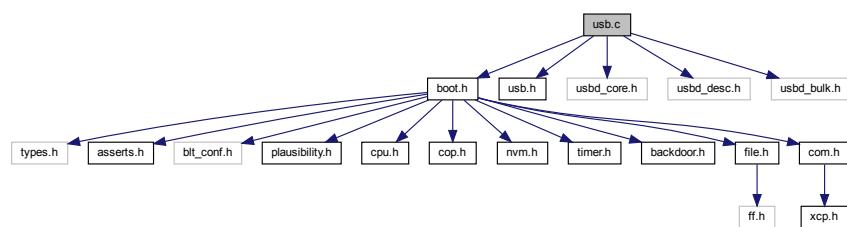
none.

7.309 usb.c File Reference

Bootloader USB communication interface source file.

```
#include "boot.h"
#include "usb.h"
#include "usbd_core.h"
#include "usbd_desc.h"
#include "usbd_bulk.h"

Include dependency graph for ARCM33_STM32U5/usb.c:
```



Data Structures

- struct **tFifoCtrl**
Structure type for fifo control.
- struct **tFifoPipe**
Structure type for a fifo pipe.

Macros

- #define **FIFO_MAX_BUFFERS** (2)
Total number of fifo buffers.
- #define **FIFO_ERR_INVALID_HANDLE** (255)
Invalid value for a fifo buffer handle.
- #define **FIFO_PIPE_SIZE** (64)
Number of bytes that fit in the fifo pipe.

Functions

- static **blt_bool UsbReceiveByte** (**blt_int8u** *data)
Receives a communication interface byte if one is present.
- static **blt_bool UsbTransmitByte** (**blt_int8u** data)
Transmits a communication interface byte.
- static void **UsbFifoMgrInit** (void)
Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.
- static **blt_int8u UsbFifoMgrCreate** (**blt_int8u** *buffer, **blt_int8u** length)
Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.
- static **blt_bool UsbFifoMgrWrite** (**blt_int8u** handle, **blt_int8u** data)
Stores data in the fifo.
- static **blt_bool UsbFifoMgrRead** (**blt_int8u** handle, **blt_int8u** *data)
Retrieves data from the fifo.
- static **blt_int8u UsbFifoMgrScan** (**blt_int8u** handle)
Returns the number of data entries currently present in the fifo.
- void **UsbInit** (void)
Initializes the USB communication interface.
- void **UsbFree** (void)
Releases the USB communication interface.
- void **UsbTransmitPacket** (**blt_int8u** *data, **blt_int8u** len)
Transmits a packet formatted for the communication interface.
- **blt_bool UsbReceivePacket** (**blt_int8u** *data, **blt_int8u** *len)
Receives a communication interface packet if one is present.
- void **UsbTransmitPipeBulkIN** (void)
Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.
- void **UsbReceivePipeBulkOUT** (**blt_int8u** epxnum)
Stores data that was received on the Bulk OUT pipe in the fifo.

Variables

- static **tFifoCtrl** **fifoCtrl** [**FIFO_MAX_BUFFERS**]
Local variable that holds the fifo control structures.
- static **tFifoCtrl** * **fifoCtrlFree**
Local pointer that points to the next free fifo control structure.
- static **tFifoPipe** **fifoPipeBulkIN**
Fifo pipe used for the bulk in endpoint.
- static **tFifoPipe** **fifoPipeBulkOUT**
Fifo pipe used for the bulk out endpoint.
- static **USBD_HandleTypeDef** **hUsbDeviceFS**
USB device handle.

7.309.1 Detailed Description

Bootloader USB communication interface source file.

7.309.2 Function Documentation

7.309.2.1 UsbFifoMgrCreate()

```
static blt_int8u UsbFifoMgrCreate (
    blt_int8u * buffer,
    blt_int8u length ) [static]
```

Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.

Parameters

| | |
|---------------|---|
| <i>buffer</i> | Pointer to the first element in the data storage fifo. |
| <i>length</i> | Maximum number of data elements that can be stored in the fifo. |

Returns

Fifo handle if successfull, or FIFO_ERR_INVALID_HANDLE.

Referenced by [UsbInit\(\)](#).

7.309.2.2 UsbFifoMgrInit()

```
static void UsbFifoMgrInit (
    void ) [static]
```

Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.

Returns

none.

Referenced by [UsbInit\(\)](#).

7.309.2.3 UsbFifoMgrRead()

```
static blt_bool UsbFifoMgrRead (
    blt_int8u handle,
    blt_int8u * data ) [static]
```

Retrieves data from the fifo.

Parameters

| | |
|---------------|---|
| <i>handle</i> | Identifies the fifo to read data from. |
| <i>data</i> | Pointer to where the read data is to be stored. |

Returns

BLT_TRUE if the data was successfully read from the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceiveByte\(\)](#), and [UsbTransmitPipeBulkIN\(\)](#).

7.309.2.4 UsbFifoMgrScan()

```
static blt_int8u UsbFifoMgrScan (
    blt_int8u handle ) [static]
```

Returns the number of data entries currently present in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo that is to be scanned. |
|---------------|--|

Returns

Number of data entries in the fifo if successful, otherwise 0.

Referenced by [UsbTransmitPipeBulkIN\(\)](#).

7.309.2.5 UsbFifoMgrWrite()

```
static blt_bool UsbFifoMgrWrite (
    blt_int8u handle,
    blt_int8u data ) [static]
```

Stores data in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo to write data to. |
| <i>data</i> | Pointer to the data that is to be written to the fifo. |

Returns

BLT_TRUE if the data was successfully stored in the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceivePipeBulkOUT\(\)](#), and [UsbTransmitByte\(\)](#).

7.309.2.6 UsbFree()

```
void UsbFree (
    void )
```

Releases the USB communication interface.

Returns

none.

7.309.2.7 UsbInit()

```
void UsbInit (
    void )
```

Initializes the USB communication interface.

Returns

none.

7.309.2.8 UsbReceiveByte()

```
static blt_bool UsbReceiveByte (
    blt_int8u * data ) [static]
```

Receives a communication interface byte if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte where the data is to be stored. |
|-------------|---|

Returns

BLT_TRUE if a byte was received, BLT_FALSE otherwise.

7.309.2.9 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.309.2.10 UsbReceivePipeBulkOUT()

```
void UsbReceivePipeBulkOUT (
    blt_int8u epnum )
```

Stores data that was received on the Bulk OUT pipe in the fifo.

Returns

none.

7.309.2.11 UsbTransmitByte()

```
static blt_bool UsbTransmitByte (
    blt_int8u data ) [static]
```

Transmits a communication interface byte.

Parameters

| | |
|-------------|--|
| <i>data</i> | Value of byte that is to be transmitted. |
|-------------|--|

Returns

BLT_TRUE if the byte was transmitted, BLT_FALSE otherwise.

Referenced by [UsbTransmitPacket\(\)](#).

7.309.2.12 UsbTransmitPacket()

```
void UsbTransmitPacket (   
    blt_int8u * data,  
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.309.2.13 UsbTransmitPipeBulkIN()

```
void UsbTransmitPipeBulkIN (   
    void )
```

Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.

Returns

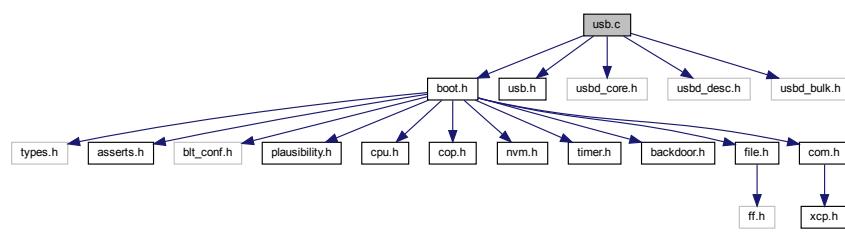
none.

7.310 usb.c File Reference

Bootloader USB communication interface source file.

```
#include "boot.h"
#include "usb.h"
#include "usbd_core.h"
#include "usbd_desc.h"
#include "usbd_bulk.h"

Include dependency graph for ARMCM3_STM32F1/usb.c:
```



Data Structures

- struct [tFifoCtrl](#)
Structure type for fifo control.
- struct [tFifoPipe](#)
Structure type for a fifo pipe.

Macros

- #define **FIFO_MAX_BUFFERS** (2)
Total number of fifo buffers.
- #define **FIFO_ERR_INVALID_HANDLE** (255)
Invalid value for a fifo buffer handle.
- #define **FIFO_PIPE_SIZE** (64)
Number of bytes that fit in the fifo pipe.

Functions

- static [blt_bool UsbReceiveByte](#) ([blt_int8u](#) *data)
Receives a communication interface byte if one is present.
- static [blt_bool UsbTransmitByte](#) ([blt_int8u](#) data)
Transmits a communication interface byte.
- static void [UsbFifoMgrInit](#) (void)
Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.
- static [blt_int8u UsbFifoMgrCreate](#) ([blt_int8u](#) *buffer, [blt_int8u](#) length)

- Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.*
- static `blt_bool UsbFifoMgrWrite (blt_int8u handle, blt_int8u data)`

Stores data in the fifo.
 - static `blt_bool UsbFifoMgrRead (blt_int8u handle, blt_int8u *data)`

Retrieves data from the fifo.
 - static `blt_int8u UsbFifoMgrScan (blt_int8u handle)`

Returns the number of data entries currently present in the fifo.
 - void `UsbInit (void)`

Initializes the USB communication interface.
 - void `UsbFree (void)`

Releases the USB communication interface.
 - void `UsbTransmitPacket (blt_int8u *data, blt_int8u len)`

Transmits a packet formatted for the communication interface.
 - `blt_bool UsbReceivePacket (blt_int8u *data, blt_int8u *len)`

Receives a communication interface packet if one is present.
 - void `UsbTransmitPipeBulkIN (void)`

Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.
 - void `UsbReceivePipeBulkOUT (blt_int8u epnum)`

Stores data that was received on the Bulk OUT pipe in the fifo.

Variables

- static `tFifoCtrl fifoCtrl [FIFO_MAX_BUFFERS]`

Local variable that holds the fifo control structures.
- static `tFifoCtrl * fifoCtrlFree`

Local pointer that points to the next free fifo control structure.
- static `tFifoPipe fifoPipeBulkIN`

Fifo pipe used for the bulk in endpoint.
- static `tFifoPipe fifoPipeBulkOUT`

Fifo pipe used for the bulk out endpoint.
- static `USBD_HandleTypeDef hUsbDeviceFS`

USB device handle.

7.310.1 Detailed Description

Bootloader USB communication interface source file.

7.310.2 Function Documentation

7.310.2.1 UsbFifoMgrCreate()

```
static blt_int8u UsbFifoMgrCreate (
    blt_int8u * buffer,
    blt_int8u length ) [static]
```

Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.

Parameters

| | |
|---------------|---|
| <i>buffer</i> | Pointer to the first element in the data storage fifo. |
| <i>length</i> | Maximum number of data elements that can be stored in the fifo. |

Returns

Fifo handle if successfull, or FIFO_ERR_INVALID_HANDLE.

Referenced by [UsbInit\(\)](#).

7.310.2.2 UsbFifoMgrInit()

```
static void UsbFifoMgrInit (
    void ) [static]
```

Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.

Returns

none.

Referenced by [UsbInit\(\)](#).

7.310.2.3 UsbFifoMgrRead()

```
static blt_bool UsbFifoMgrRead (
    blt_int8u handle,
    blt_int8u * data ) [static]
```

Retrieves data from the fifo.

Parameters

| | |
|---------------|---|
| <i>handle</i> | Identifies the fifo to read data from. |
| <i>data</i> | Pointer to where the read data is to be stored. |

Returns

BLT_TRUE if the data was successfully read from the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceiveByte\(\)](#), and [UsbTransmitPipeBulkIN\(\)](#).

7.310.2.4 UsbFifoMgrScan()

```
static blt_int8u UsbFifoMgrScan (
    blt_int8u handle ) [static]
```

Returns the number of data entries currently present in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo that is to be scanned. |
|---------------|--|

Returns

Number of data entries in the fifo if successful, otherwise 0.

Referenced by [UsbTransmitPipeBulkIN\(\)](#).

7.310.2.5 UsbFifoMgrWrite()

```
static blt_bool UsbFifoMgrWrite (
    blt_int8u handle,
    blt_int8u data ) [static]
```

Stores data in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo to write data to. |
| <i>data</i> | Pointer to the data that is to be written to the fifo. |

Returns

BLT_TRUE if the data was successfully stored in the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceivePipeBulkOUT\(\)](#), and [UsbTransmitByte\(\)](#).

7.310.2.6 UsbFree()

```
void UsbFree (
    void )
```

Releases the USB communication interface.

Returns

none.

7.310.2.7 UsbInit()

```
void UsbInit (
    void )
```

Initializes the USB communication interface.

Returns

none.

7.310.2.8 UsbReceiveByte()

```
static blt_bool UsbReceiveByte (
    blt_int8u * data ) [static]
```

Receives a communication interface byte if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte where the data is to be stored. |
|-------------|---|

Returns

BLT_TRUE if a byte was received, BLT_FALSE otherwise.

7.310.2.9 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.310.2.10 UsbReceivePipeBulkOUT()

```
void UsbReceivePipeBulkOUT (
    blt_int8u eppnum )
```

Stores data that was received on the Bulk OUT pipe in the fifo.

Returns

none.

7.310.2.11 UsbTransmitByte()

```
static blt_bool UsbTransmitByte (
    blt_int8u data ) [static]
```

Transmits a communication interface byte.

Parameters

| | |
|-------------|--|
| <i>data</i> | Value of byte that is to be transmitted. |
|-------------|--|

Returns

BLT_TRUE if the byte was transmitted, BLT_FALSE otherwise.

Referenced by [UsbTransmitPacket\(\)](#).

7.310.2.12 UsbTransmitPacket()

```
void UsbTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.310.2.13 UsbTransmitPipeBulkIN()

```
void UsbTransmitPipeBulkIN (
    void )
```

Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.

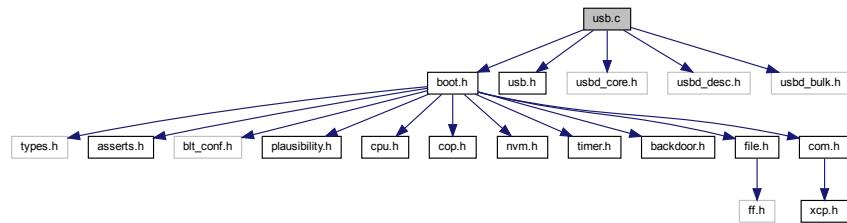
Returns

none.

7.311 usb.c File Reference

Bootloader USB communication interface source file.

```
#include "boot.h"
#include "usb.h"
#include "usbd_core.h"
#include "usbd_desc.h"
#include "usbd_bulk.h"
Include dependency graph for ARMCM3_STM32F2/usb.c:
```



Data Structures

- struct [tFifoCtrl](#)
Structure type for fifo control.
- struct [tFifoPipe](#)
Structure type for a fifo pipe.

Macros

- #define **FIFO_MAX_BUFFERS** (2)
Total number of fifo buffers.
- #define **FIFO_ERR_INVALID_HANDLE** (255)
Invalid value for a fifo buffer handle.
- #define **FIFO_PIPE_SIZE** (64)
Number of bytes that fit in the fifo pipe.

Functions

- static `blt_bool UsbReceiveByte (blt_int8u *data)`
Receives a communication interface byte if one is present.
- static `blt_bool UsbTransmitByte (blt_int8u data)`
Transmits a communication interface byte.
- static void `UsbFifoMgrInit (void)`
Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into `fifoCtrl[]`. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in `fifoCtrl[]` this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.
- static `blt_int8u UsbFifoMgrCreate (blt_int8u *buffer, blt_int8u length)`
Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.
- static `blt_bool UsbFifoMgrWrite (blt_int8u handle, blt_int8u data)`
Stores data in the fifo.
- static `blt_bool UsbFifoMgrRead (blt_int8u handle, blt_int8u *data)`
Retrieves data from the fifo.
- static `blt_int8u UsbFifoMgrScan (blt_int8u handle)`
Returns the number of data entries currently present in the fifo.
- void `UsbInit (void)`
Initializes the USB communication interface.
- void `UsbFree (void)`
Releases the USB communication interface.
- void `UsbTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool UsbReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.
- void `UsbTransmitPipeBulkIN (void)`
Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.
- void `UsbReceivePipeBulkOUT (blt_int8u epnum)`
Stores data that was received on the Bulk OUT pipe in the fifo.

Variables

- static `tFifoCtrl fifoCtrl [FIFO_MAX_BUFFERS]`
Local variable that holds the fifo control structures.
- static `tFifoCtrl * fifoCtrlFree`
Local pointer that points to the next free fifo control structure.
- static `tFifoPipe fifoPipeBulkIN`
Fifo pipe used for the bulk in endpoint.
- static `tFifoPipe fifoPipeBulkOUT`
Fifo pipe used for the bulk out endpoint.
- static `USBD_HandleTypeDef hUsbDeviceFS`
USB device handle.

7.311.1 Detailed Description

Bootloader USB communication interface source file.

7.311.2 Function Documentation

7.311.2.1 UsbFifoMgrCreate()

```
static blt_int8u UsbFifoMgrCreate (
    blt_int8u * buffer,
    blt_int8u length ) [static]
```

Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.

Parameters

| | |
|---------------|---|
| <i>buffer</i> | Pointer to the first element in the data storage fifo. |
| <i>length</i> | Maximum number of data elements that can be stored in the fifo. |

Returns

Fifo handle if successfull, or FIFO_ERR_INVALID_HANDLE.

Referenced by [UsbInit\(\)](#).

7.311.2.2 UsbFifoMgrInit()

```
static void UsbFifoMgrInit (
    void ) [static]
```

Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.

Returns

none.

Referenced by [UsbInit\(\)](#).

7.311.2.3 UsbFifoMgrRead()

```
static blt_bool UsbFifoMgrRead (
    blt_int8u handle,
    blt_int8u * data ) [static]
```

Retrieves data from the fifo.

Parameters

| | |
|---------------|---|
| <i>handle</i> | Identifies the fifo to read data from. |
| <i>data</i> | Pointer to where the read data is to be stored. |

Returns

BLT_TRUE if the data was successfully read from the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceiveByte\(\)](#), and [UsbTransmitPipeBulkIN\(\)](#).

7.311.2.4 UsbFifoMgrScan()

```
static blt_int8u UsbFifoMgrScan (
    blt_int8u handle ) [static]
```

Returns the number of data entries currently present in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo that is to be scanned. |
|---------------|--|

Returns

Number of data entries in the fifo if successful, otherwise 0.

Referenced by [UsbTransmitPipeBulkIN\(\)](#).

7.311.2.5 UsbFifoMgrWrite()

```
static blt_bool UsbFifoMgrWrite (
    blt_int8u handle,
    blt_int8u data ) [static]
```

Stores data in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo to write data to. |
| <i>data</i> | Pointer to the data that is to be written to the fifo. |

Returns

BLT_TRUE if the data was successfully stored in the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceivePipeBulkOUT\(\)](#), and [UsbTransmitByte\(\)](#).

7.311.2.6 UsbFree()

```
void UsbFree (
    void )
```

Releases the USB communication interface.

Returns

none.

7.311.2.7 UsbInit()

```
void UsbInit (
    void )
```

Initializes the USB communication interface.

Returns

none.

7.311.2.8 UsbReceiveByte()

```
static blt_bool UsbReceiveByte (
    blt_int8u * data ) [static]
```

Receives a communication interface byte if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte where the data is to be stored. |
|-------------|---|

Returns

BLT_TRUE if a byte was received, BLT_FALSE otherwise.

7.311.2.9 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.311.2.10 UsbReceivePipeBulkOUT()

```
void UsbReceivePipeBulkOUT (
    blt_int8u epnum )
```

Stores data that was received on the Bulk OUT pipe in the fifo.

Returns

none.

7.311.2.11 UsbTransmitByte()

```
static blt_bool UsbTransmitByte (
    blt_int8u data ) [static]
```

Transmits a communication interface byte.

Parameters

| | |
|-------------|--|
| <i>data</i> | Value of byte that is to be transmitted. |
|-------------|--|

Returns

BLT_TRUE if the byte was transmitted, BLT_FALSE otherwise.

Referenced by [UsbTransmitPacket\(\)](#).

7.311.2.12 UsbTransmitPacket()

```
void UsbTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.311.2.13 UsbTransmitPipeBulkIN()

```
void UsbTransmitPipeBulkIN (
    void )
```

Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.

Returns

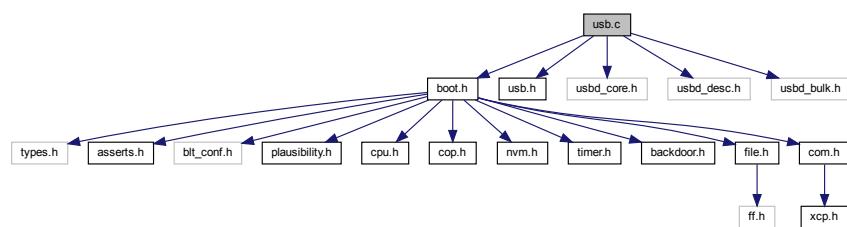
none.

7.312 usb.c File Reference

Bootloader USB communication interface source file.

```
#include "boot.h"
#include "usb.h"
#include "usbd_core.h"
#include "usbd_desc.h"
#include "usbd_bulk.h"

Include dependency graph for ARMCM4_STM32F3/usb.c:
```



Data Structures

- struct **tFifoCtrl**
Structure type for fifo control.
- struct **tFifoPipe**
Structure type for a fifo pipe.

Macros

- #define **FIFO_MAX_BUFFERS** (2)
Total number of fifo buffers.
- #define **FIFO_ERR_INVALID_HANDLE** (255)
Invalid value for a fifo buffer handle.
- #define **FIFO_PIPE_SIZE** (64)
Number of bytes that fit in the fifo pipe.

Functions

- static **blt_bool UsbReceiveByte** (**blt_int8u** *data)
Receives a communication interface byte if one is present.
- static **blt_bool UsbTransmitByte** (**blt_int8u** data)
Transmits a communication interface byte.
- static void **UsbFifoMgrInit** (void)
Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.
- static **blt_int8u UsbFifoMgrCreate** (**blt_int8u** *buffer, **blt_int8u** length)
Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.
- static **blt_bool UsbFifoMgrWrite** (**blt_int8u** handle, **blt_int8u** data)
Stores data in the fifo.
- static **blt_bool UsbFifoMgrRead** (**blt_int8u** handle, **blt_int8u** *data)
Retrieves data from the fifo.
- static **blt_int8u UsbFifoMgrScan** (**blt_int8u** handle)
Returns the number of data entries currently present in the fifo.
- void **UsbInit** (void)
Initializes the USB communication interface.
- void **UsbFree** (void)
Releases the USB communication interface.
- void **UsbTransmitPacket** (**blt_int8u** *data, **blt_int8u** len)
Transmits a packet formatted for the communication interface.
- **blt_bool UsbReceivePacket** (**blt_int8u** *data, **blt_int8u** *len)
Receives a communication interface packet if one is present.
- void **UsbTransmitPipeBulkIN** (void)
Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.
- void **UsbReceivePipeBulkOUT** (**blt_int8u** epxnum)
Stores data that was received on the Bulk OUT pipe in the fifo.

Variables

- static **tFifoCtrl** **fifoCtrl** [**FIFO_MAX_BUFFERS**]
Local variable that holds the fifo control structures.
- static **tFifoCtrl** * **fifoCtrlFree**
Local pointer that points to the next free fifo control structure.
- static **tFifoPipe** **fifoPipeBulkIN**
Fifo pipe used for the bulk in endpoint.
- static **tFifoPipe** **fifoPipeBulkOUT**
Fifo pipe used for the bulk out endpoint.
- static **USBD_HandleTypeDef** **hUsbDeviceFS**
USB device handle.

7.312.1 Detailed Description

Bootloader USB communication interface source file.

7.312.2 Function Documentation

7.312.2.1 UsbFifoMgrCreate()

```
static blt_int8u UsbFifoMgrCreate (
    blt_int8u * buffer,
    blt_int8u length ) [static]
```

Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.

Parameters

| | |
|---------------|---|
| <i>buffer</i> | Pointer to the first element in the data storage fifo. |
| <i>length</i> | Maximum number of data elements that can be stored in the fifo. |

Returns

Fifo handle if successfull, or FIFO_ERR_INVALID_HANDLE.

Referenced by [UsbInit\(\)](#).

7.312.2.2 UsbFifoMgrInit()

```
static void UsbFifoMgrInit (
    void ) [static]
```

Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.

Returns

none.

Referenced by [UsbInit\(\)](#).

7.312.2.3 UsbFifoMgrRead()

```
static blt_bool UsbFifoMgrRead (
    blt_int8u handle,
    blt_int8u * data ) [static]
```

Retrieves data from the fifo.

Parameters

| | |
|---------------|---|
| <i>handle</i> | Identifies the fifo to read data from. |
| <i>data</i> | Pointer to where the read data is to be stored. |

Returns

BLT_TRUE if the data was successfully read from the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceiveByte\(\)](#), and [UsbTransmitPipeBulkIN\(\)](#).

7.312.2.4 UsbFifoMgrScan()

```
static blt_int8u UsbFifoMgrScan (
    blt_int8u handle ) [static]
```

Returns the number of data entries currently present in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo that is to be scanned. |
|---------------|--|

Returns

Number of data entries in the fifo if successful, otherwise 0.

Referenced by [UsbTransmitPipeBulkIN\(\)](#).

7.312.2.5 UsbFifoMgrWrite()

```
static blt_bool UsbFifoMgrWrite (
    blt_int8u handle,
    blt_int8u data ) [static]
```

Stores data in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo to write data to. |
| <i>data</i> | Pointer to the data that is to be written to the fifo. |

Returns

BLT_TRUE if the data was successfully stored in the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceivePipeBulkOUT\(\)](#), and [UsbTransmitByte\(\)](#).

7.312.2.6 UsbFree()

```
void UsbFree (
    void )
```

Releases the USB communication interface.

Returns

none.

7.312.2.7 UsbInit()

```
void UsbInit (
    void )
```

Initializes the USB communication interface.

Returns

none.

7.312.2.8 UsbReceiveByte()

```
static blt_bool UsbReceiveByte (
    blt_int8u * data ) [static]
```

Receives a communication interface byte if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte where the data is to be stored. |
|-------------|---|

Returns

BLT_TRUE if a byte was received, BLT_FALSE otherwise.

7.312.2.9 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.312.2.10 UsbReceivePipeBulkOUT()

```
void UsbReceivePipeBulkOUT (
    blt_int8u epnum )
```

Stores data that was received on the Bulk OUT pipe in the fifo.

Returns

none.

7.312.2.11 UsbTransmitByte()

```
static blt_bool UsbTransmitByte (
    blt_int8u data ) [static]
```

Transmits a communication interface byte.

Parameters

| | |
|-------------|--|
| <i>data</i> | Value of byte that is to be transmitted. |
|-------------|--|

Returns

BLT_TRUE if the byte was transmitted, BLT_FALSE otherwise.

Referenced by [UsbTransmitPacket\(\)](#).

7.312.2.12 UsbTransmitPacket()

```
void UsbTransmitPacket (   
    blt_int8u * data,  
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.312.2.13 UsbTransmitPipeBulkIN()

```
void UsbTransmitPipeBulkIN (   
    void )
```

Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.

Returns

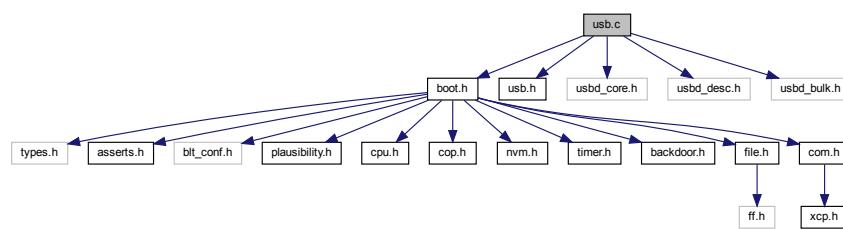
none.

7.313 usb.c File Reference

Bootloader USB communication interface source file.

```
#include "boot.h"
#include "usb.h"
#include "usbd_core.h"
#include "usbd_desc.h"
#include "usbd_bulk.h"

Include dependency graph for ARMCM4_STM32F4/usb.c:
```



Data Structures

- struct [tFifoCtrl](#)
Structure type for fifo control.
- struct [tFifoPipe](#)
Structure type for a fifo pipe.

Macros

- #define **FIFO_MAX_BUFFERS** (2)
Total number of fifo buffers.
- #define **FIFO_ERR_INVALID_HANDLE** (255)
Invalid value for a fifo buffer handle.
- #define **FIFO_PIPE_SIZE** (64)
Number of bytes that fit in the fifo pipe.

Functions

- static [blt_bool UsbReceiveByte](#) ([blt_int8u](#) *data)
Receives a communication interface byte if one is present.
- static [blt_bool UsbTransmitByte](#) ([blt_int8u](#) data)
Transmits a communication interface byte.
- static void [UsbFifoMgrInit](#) (void)
Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.
- static [blt_int8u UsbFifoMgrCreate](#) ([blt_int8u](#) *buffer, [blt_int8u](#) length)

- Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.*
- static `blt_bool UsbFifoMgrWrite (blt_int8u handle, blt_int8u data)`

Stores data in the fifo.
 - static `blt_bool UsbFifoMgrRead (blt_int8u handle, blt_int8u *data)`

Retrieves data from the fifo.
 - static `blt_int8u UsbFifoMgrScan (blt_int8u handle)`

Returns the number of data entries currently present in the fifo.
 - void `UsbInit (void)`

Initializes the USB communication interface.
 - void `UsbFree (void)`

Releases the USB communication interface.
 - void `UsbTransmitPacket (blt_int8u *data, blt_int8u len)`

Transmits a packet formatted for the communication interface.
 - `blt_bool UsbReceivePacket (blt_int8u *data, blt_int8u *len)`

Receives a communication interface packet if one is present.
 - void `UsbTransmitPipeBulkIN (void)`

Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.
 - void `UsbReceivePipeBulkOUT (blt_int8u epnum)`

Stores data that was received on the Bulk OUT pipe in the fifo.

Variables

- static `tFifoCtrl fifoCtrl [FIFO_MAX_BUFFERS]`

Local variable that holds the fifo control structures.
- static `tFifoCtrl * fifoCtrlFree`

Local pointer that points to the next free fifo control structure.
- static `tFifoPipe fifoPipeBulkIN`

Fifo pipe used for the bulk in endpoint.
- static `tFifoPipe fifoPipeBulkOUT`

Fifo pipe used for the bulk out endpoint.
- static `USBD_HandleTypeDef hUsbDeviceFS`

USB device handle.

7.313.1 Detailed Description

Bootloader USB communication interface source file.

7.313.2 Function Documentation

7.313.2.1 UsbFifoMgrCreate()

```
static blt_int8u UsbFifoMgrCreate (
    blt_int8u * buffer,
    blt_int8u length ) [static]
```

Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.

Parameters

| | |
|---------------|---|
| <i>buffer</i> | Pointer to the first element in the data storage fifo. |
| <i>length</i> | Maximum number of data elements that can be stored in the fifo. |

Returns

Fifo handle if successfull, or FIFO_ERR_INVALID_HANDLE.

Referenced by [UsbInit\(\)](#).

7.313.2.2 UsbFifoMgrInit()

```
static void UsbFifoMgrInit (
    void ) [static]
```

Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.

Returns

none.

Referenced by [UsbInit\(\)](#).

7.313.2.3 UsbFifoMgrRead()

```
static blt_bool UsbFifoMgrRead (
    blt_int8u handle,
    blt_int8u * data ) [static]
```

Retrieves data from the fifo.

Parameters

| | |
|---------------|---|
| <i>handle</i> | Identifies the fifo to read data from. |
| <i>data</i> | Pointer to where the read data is to be stored. |

Returns

BLT_TRUE if the data was successfully read from the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceiveByte\(\)](#), and [UsbTransmitPipeBulkIN\(\)](#).

7.313.2.4 UsbFifoMgrScan()

```
static blt_int8u UsbFifoMgrScan (
    blt_int8u handle ) [static]
```

Returns the number of data entries currently present in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo that is to be scanned. |
|---------------|--|

Returns

Number of data entries in the fifo if successful, otherwise 0.

Referenced by [UsbTransmitPipeBulkIN\(\)](#).

7.313.2.5 UsbFifoMgrWrite()

```
static blt_bool UsbFifoMgrWrite (
    blt_int8u handle,
    blt_int8u data ) [static]
```

Stores data in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo to write data to. |
| <i>data</i> | Pointer to the data that is to be written to the fifo. |

Returns

BLT_TRUE if the data was successfully stored in the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceivePipeBulkOUT\(\)](#), and [UsbTransmitByte\(\)](#).

7.313.2.6 UsbFree()

```
void UsbFree (
    void )
```

Releases the USB communication interface.

Returns

none.

7.313.2.7 UsbInit()

```
void UsbInit (
    void  )
```

Initializes the USB communication interface.

Returns

none.

7.313.2.8 UsbReceiveByte()

```
static blt_bool UsbReceiveByte (
    blt_int8u * data ) [static]
```

Receives a communication interface byte if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte where the data is to be stored. |
|-------------|---|

Returns

BLT_TRUE if a byte was received, BLT_FALSE otherwise.

7.313.2.9 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.313.2.10 UsbReceivePipeBulkOUT()

```
void UsbReceivePipeBulkOUT (
    blt_int8u eppnum )
```

Stores data that was received on the Bulk OUT pipe in the fifo.

Returns

none.

7.313.2.11 UsbTransmitByte()

```
static blt_bool UsbTransmitByte (
    blt_int8u data ) [static]
```

Transmits a communication interface byte.

Parameters

| | |
|-------------|--|
| <i>data</i> | Value of byte that is to be transmitted. |
|-------------|--|

Returns

BLT_TRUE if the byte was transmitted, BLT_FALSE otherwise.

Referenced by [UsbTransmitPacket\(\)](#).

7.313.2.12 UsbTransmitPacket()

```
void UsbTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.313.2.13 UsbTransmitPipeBulkIN()

```
void UsbTransmitPipeBulkIN (
    void )
```

Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.

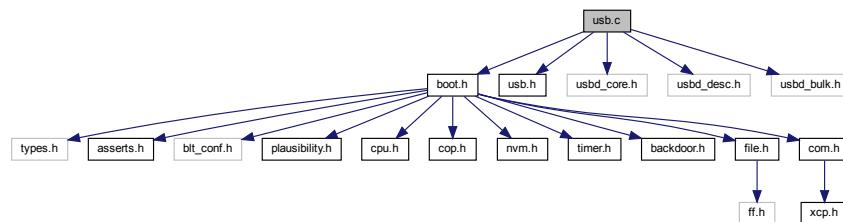
Returns

none.

7.314 usb.c File Reference

Bootloader USB communication interface source file.

```
#include "boot.h"
#include "usb.h"
#include "usbd_core.h"
#include "usbd_desc.h"
#include "usbd_bulk.h"
Include dependency graph for ARMCM4_STM32L4/usb.c:
```



Data Structures

- struct `tFifoCtrl`
Structure type for fifo control.
- struct `tFifoPipe`
Structure type for a fifo pipe.

Macros

- #define `FIFO_MAX_BUFFERS` (2)
Total number of fifo buffers.
- #define `FIFO_ERR_INVALID_HANDLE` (255)
Invalid value for a fifo buffer handle.
- #define `FIFO_PIPE_SIZE` (64)
Number of bytes that fit in the fifo pipe.

Functions

- static `blt_bool UsbReceiveByte (blt_int8u *data)`
Receives a communication interface byte if one is present.
- static `blt_bool UsbTransmitByte (blt_int8u data)`
Transmits a communication interface byte.
- static void `UsbFifoMgrInit (void)`
Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into `fifoCtrl[]`. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in `fifoCtrl[]` this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.
- static `blt_int8u UsbFifoMgrCreate (blt_int8u *buffer, blt_int8u length)`
Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.
- static `blt_bool UsbFifoMgrWrite (blt_int8u handle, blt_int8u data)`
Stores data in the fifo.
- static `blt_bool UsbFifoMgrRead (blt_int8u handle, blt_int8u *data)`
Retrieves data from the fifo.
- static `blt_int8u UsbFifoMgrScan (blt_int8u handle)`
Returns the number of data entries currently present in the fifo.
- void `UsbInit (void)`
Initializes the USB communication interface.
- void `UsbFree (void)`
Releases the USB communication interface.
- void `UsbTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool UsbReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.
- void `UsbTransmitPipeBulkIN (void)`
Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.
- void `UsbReceivePipeBulkOUT (blt_int8u epnum)`
Stores data that was received on the Bulk OUT pipe in the fifo.

Variables

- static `tFifoCtrl fifoCtrl [FIFO_MAX_BUFFERS]`
Local variable that holds the fifo control structures.
- static `tFifoCtrl * fifoCtrlFree`
Local pointer that points to the next free fifo control structure.
- static `tFifoPipe fifoPipeBulkIN`
Fifo pipe used for the bulk in endpoint.
- static `tFifoPipe fifoPipeBulkOUT`
Fifo pipe used for the bulk out endpoint.
- static `USBD_HandleTypeDef hUsbDeviceFS`
USB device handle.

7.314.1 Detailed Description

Bootloader USB communication interface source file.

7.314.2 Function Documentation

7.314.2.1 UsbFifoMgrCreate()

```
static blt_int8u UsbFifoMgrCreate (
    blt_int8u * buffer,
    blt_int8u length ) [static]
```

Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.

Parameters

| | |
|---------------|---|
| <i>buffer</i> | Pointer to the first element in the data storage fifo. |
| <i>length</i> | Maximum number of data elements that can be stored in the fifo. |

Returns

Fifo handle if successfull, or FIFO_ERR_INVALID_HANDLE.

Referenced by [UsbInit\(\)](#).

7.314.2.2 UsbFifoMgrInit()

```
static void UsbFifoMgrInit (
    void ) [static]
```

Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.

Returns

none.

Referenced by [UsbInit\(\)](#).

7.314.2.3 UsbFifoMgrRead()

```
static blt_bool UsbFifoMgrRead (
    blt_int8u handle,
    blt_int8u * data ) [static]
```

Retrieves data from the fifo.

Parameters

| | |
|---------------|---|
| <i>handle</i> | Identifies the fifo to read data from. |
| <i>data</i> | Pointer to where the read data is to be stored. |

Returns

BLT_TRUE if the data was successfully read from the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceiveByte\(\)](#), and [UsbTransmitPipeBulkIN\(\)](#).

7.314.2.4 UsbFifoMgrScan()

```
static blt_int8u UsbFifoMgrScan (
    blt_int8u handle ) [static]
```

Returns the number of data entries currently present in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo that is to be scanned. |
|---------------|--|

Returns

Number of data entries in the fifo if successful, otherwise 0.

Referenced by [UsbTransmitPipeBulkIN\(\)](#).

7.314.2.5 UsbFifoMgrWrite()

```
static blt_bool UsbFifoMgrWrite (
    blt_int8u handle,
    blt_int8u data ) [static]
```

Stores data in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo to write data to. |
| <i>data</i> | Pointer to the data that is to be written to the fifo. |

Returns

BLT_TRUE if the data was successfully stored in the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceivePipeBulkOUT\(\)](#), and [UsbTransmitByte\(\)](#).

7.314.2.6 UsbFree()

```
void UsbFree (
    void )
```

Releases the USB communication interface.

Returns

none.

7.314.2.7 UsbInit()

```
void UsbInit (
    void )
```

Initializes the USB communication interface.

Returns

none.

7.314.2.8 UsbReceiveByte()

```
static blt_bool UsbReceiveByte (
    blt_int8u * data ) [static]
```

Receives a communication interface byte if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte where the data is to be stored. |
|-------------|---|

Returns

BLT_TRUE if a byte was received, BLT_FALSE otherwise.

7.314.2.9 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.314.2.10 UsbReceivePipeBulkOUT()

```
void UsbReceivePipeBulkOUT (
    blt_int8u epnum )
```

Stores data that was received on the Bulk OUT pipe in the fifo.

Returns

none.

7.314.2.11 UsbTransmitByte()

```
static blt_bool UsbTransmitByte (
    blt_int8u data ) [static]
```

Transmits a communication interface byte.

Parameters

| | |
|-------------|--|
| <i>data</i> | Value of byte that is to be transmitted. |
|-------------|--|

Returns

BLT_TRUE if the byte was transmitted, BLT_FALSE otherwise.

Referenced by [UsbTransmitPacket\(\)](#).

7.314.2.12 UsbTransmitPacket()

```
void UsbTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.314.2.13 UsbTransmitPipeBulkIN()

```
void UsbTransmitPipeBulkIN (
    void )
```

Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.

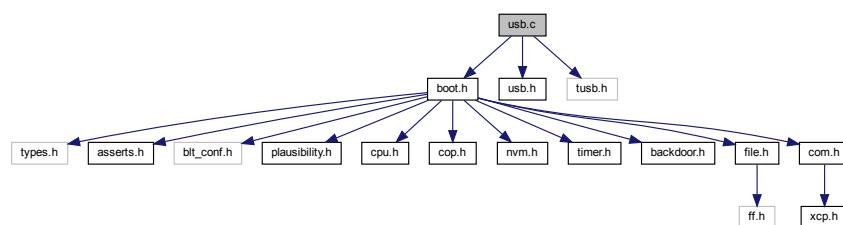
Returns

none.

7.315 usb.c File Reference

Bootloader USB communication interface source file.

```
#include "boot.h"
#include "usb.h"
#include "tusb.h"
Include dependency graph for ARMCM4_TM4C/usb.c:
```



Functions

- static `blt_bool UsbReceiveByte (blt_int8u *data)`
Receives a communication interface byte if one is present.
- void `UsbInit (void)`
Initializes the USB communication interface.
- void `UsbFree (void)`
Releases the USB communication interface.
- void `UsbTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool UsbReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.
- void `tud_mount_cb (void)`
TinyUSB stack callback invoked then the USB device is mounted.
- void `tud_umount_cb (void)`
TinyUSB stack callback invoked then the USB device is unmounted.
- void `tud_suspend_cb (bool remote_wakeup_en)`
TinyUSB stack callback invoked then the USB bus is suspended. Within 7ms the device must draw an average of current less than 2.5 mA from the bus.
- void `tud_resume_cb (void)`
TinyUSB stack callback invoked then the USB bus is resumed.

7.315.1 Detailed Description

Bootloader USB communication interface source file.

7.315.2 Function Documentation

7.315.2.1 tud_suspend_cb()

```
void tud_suspend_cb (
    bool remote_wakeup_en )
```

TinyUSB stack callback invoked then the USB bus is suspended. Within 7ms the device must draw an average of current less than 2.5 mA from the bus.

Parameters

| | |
|-------------------------------|--|
| <code>remote_wakeup_en</code> | True if the host allows us to perform a remote wakeup. |
|-------------------------------|--|

7.315.2.2 UsbFree()

```
void UsbFree (
    void  )
```

Releases the USB communication interface.

Returns

none.

7.315.2.3 UsbInit()

```
void UsbInit (
    void  )
```

Initializes the USB communication interface.

Returns

none.

7.315.2.4 UsbReceiveByte()

```
static blt_bool UsbReceiveByte (
    blt_int8u * data )  [static]
```

Receives a communication interface byte if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte where the data is to be stored. |
|-------------|---|

Returns

BLT_TRUE if a byte was received, BLT_FALSE otherwise.

7.315.2.5 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.315.2.6 UsbTransmitPacket()

```
void UsbTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

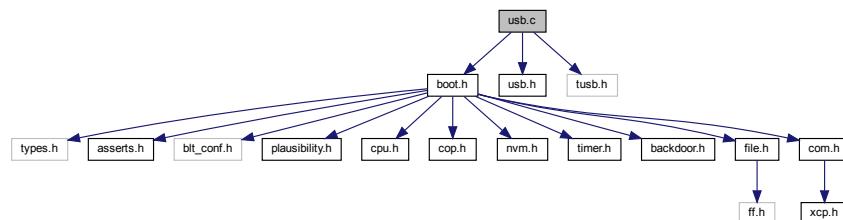
Returns

none.

7.316 usb.c File Reference

Bootloader USB communication interface source file.

```
#include "boot.h"
#include "usb.h"
#include "tusb.h"
Include dependency graph for ARMCM4_XMC4/usb.c:
```



Functions

- static `blt_bool UsbReceiveByte (blt_int8u *data)`
Receives a communication interface byte if one is present.
- void `UsbInit (void)`
Initializes the USB communication interface.
- void `UsbFree (void)`
Releases the USB communication interface.
- void `UsbTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool UsbReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.
- void `tud_mount_cb (void)`
TinyUSB stack callback invoked then the USB device is mounted.
- void `tud_umount_cb (void)`
TinyUSB stack callback invoked then the USB device is unmounted.
- void `tud_suspend_cb (bool remote_wakeup_en)`
TinyUSB stack callback invoked then the USB bus is suspended. Within 7ms the device must draw an average of current less than 2.5 mA from the bus.
- void `tud_resume_cb (void)`
TinyUSB stack callback invoked then the USB bus is resumed.

7.316.1 Detailed Description

Bootloader USB communication interface source file.

7.316.2 Function Documentation

7.316.2.1 tud_suspend_cb()

```
void tud_suspend_cb (
    bool remote_wakeup_en )
```

TinyUSB stack callback invoked then the USB bus is suspended. Within 7ms the device must draw an average of current less than 2.5 mA from the bus.

Parameters

| | |
|-------------------------------|--|
| <code>remote_wakeup_en</code> | True if the host allows us to perform a remote wakeup. |
|-------------------------------|--|

7.316.2.2 UsbFree()

```
void UsbFree (
    void  )
```

Releases the USB communication interface.

Returns

none.

7.316.2.3 UsbInit()

```
void UsbInit (
    void  )
```

Initializes the USB communication interface.

Returns

none.

7.316.2.4 UsbReceiveByte()

```
static blt_bool UsbReceiveByte (
    blt_int8u * data )  [static]
```

Receives a communication interface byte if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte where the data is to be stored. |
|-------------|---|

Returns

BLT_TRUE if a byte was received, BLT_FALSE otherwise.

7.316.2.5 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.316.2.6 UsbTransmitPacket()

```
void UsbTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

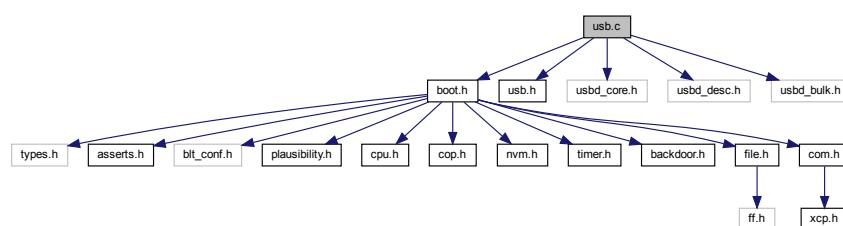
Returns

none.

7.317 usb.c File Reference

Bootloader USB communication interface source file.

```
#include "boot.h"
#include "usb.h"
#include "usbd_core.h"
#include "usbd_desc.h"
#include "usbd_bulk.h"
Include dependency graph for ARMCM7_STM32F7/usb.c:
```



Data Structures

- struct **tFifoCtrl**
Structure type for fifo control.
- struct **tFifoPipe**
Structure type for a fifo pipe.

Macros

- #define **FIFO_MAX_BUFFERS** (2)
Total number of fifo buffers.
- #define **FIFO_ERR_INVALID_HANDLE** (255)
Invalid value for a fifo buffer handle.
- #define **FIFO_PIPE_SIZE** (64)
Number of bytes that fit in the fifo pipe.

Functions

- static **blt_bool UsbReceiveByte** (**blt_int8u** *data)
Receives a communication interface byte if one is present.
- static **blt_bool UsbTransmitByte** (**blt_int8u** data)
Transmits a communication interface byte.
- static void **UsbFifoMgrInit** (void)
Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.
- static **blt_int8u UsbFifoMgrCreate** (**blt_int8u** *buffer, **blt_int8u** length)
Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.
- static **blt_bool UsbFifoMgrWrite** (**blt_int8u** handle, **blt_int8u** data)
Stores data in the fifo.
- static **blt_bool UsbFifoMgrRead** (**blt_int8u** handle, **blt_int8u** *data)
Retrieves data from the fifo.
- static **blt_int8u UsbFifoMgrScan** (**blt_int8u** handle)
Returns the number of data entries currently present in the fifo.
- void **UsbInit** (void)
Initializes the USB communication interface.
- void **UsbFree** (void)
Releases the USB communication interface.
- void **UsbTransmitPacket** (**blt_int8u** *data, **blt_int8u** len)
Transmits a packet formatted for the communication interface.
- **blt_bool UsbReceivePacket** (**blt_int8u** *data, **blt_int8u** *len)
Receives a communication interface packet if one is present.
- void **UsbTransmitPipeBulkIN** (void)
Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.
- void **UsbReceivePipeBulkOUT** (**blt_int8u** epxnum)
Stores data that was received on the Bulk OUT pipe in the fifo.

Variables

- static tFifoCtrl **fifoCtrl** [FIFO_MAX_BUFFERS]
Local variable that holds the fifo control structures.
- static tFifoCtrl * **fifoCtrlFree**
Local pointer that points to the next free fifo control structure.
- static tFifoPipe **fifoPipeBulkIN**
Fifo pipe used for the bulk in endpoint.
- static tFifoPipe **fifoPipeBulkOUT**
Fifo pipe used for the bulk out endpoint.
- static USBD_HandleTypeDef **hUsbDeviceFS**
USB device handle.

7.317.1 Detailed Description

Bootloader USB communication interface source file.

7.317.2 Function Documentation

7.317.2.1 UsbFifoMgrCreate()

```
static blt_int8u UsbFifoMgrCreate (
    blt_int8u * buffer,
    blt_int8u length ) [static]
```

Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.

Parameters

| | |
|---------------|---|
| <i>buffer</i> | Pointer to the first element in the data storage fifo. |
| <i>length</i> | Maximum number of data elements that can be stored in the fifo. |

Returns

Fifo handle if successfull, or FIFO_ERR_INVALID_HANDLE.

Referenced by [UsbInit\(\)](#).

7.317.2.2 UsbFifoMgrInit()

```
static void UsbFifoMgrInit (
    void ) [static]
```

Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.

Returns

none.

Referenced by [UsbInit\(\)](#).

7.317.2.3 UsbFifoMgrRead()

```
static blt_bool UsbFifoMgrRead (
    blt_int8u handle,
    blt_int8u * data ) [static]
```

Retrieves data from the fifo.

Parameters

| | |
|---------------|---|
| <i>handle</i> | Identifies the fifo to read data from. |
| <i>data</i> | Pointer to where the read data is to be stored. |

Returns

BLT_TRUE if the data was successfully read from the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceiveByte\(\)](#), and [UsbTransmitPipeBulkIN\(\)](#).

7.317.2.4 UsbFifoMgrScan()

```
static blt_int8u UsbFifoMgrScan (
    blt_int8u handle ) [static]
```

Returns the number of data entries currently present in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo that is to be scanned. |
|---------------|--|

Returns

Number of data entries in the fifo if successful, otherwise 0.

Referenced by [UsbTransmitPipeBulkIN\(\)](#).

7.317.2.5 UsbFifoMgrWrite()

```
static blt_bool UsbFifoMgrWrite (
    blt_int8u handle,
    blt_int8u data ) [static]
```

Stores data in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo to write data to. |
| <i>data</i> | Pointer to the data that is to be written to the fifo. |

Returns

BLT_TRUE if the data was successfully stored in the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceivePipeBulkOUT\(\)](#), and [UsbTransmitByte\(\)](#).

7.317.2.6 UsbFree()

```
void UsbFree (
    void )
```

Releases the USB communication interface.

Returns

none.

7.317.2.7 UsbInit()

```
void UsbInit (
    void )
```

Initializes the USB communication interface.

Returns

none.

7.317.2.8 UsbReceiveByte()

```
static blt_bool UsbReceiveByte (
    blt_int8u * data ) [static]
```

Receives a communication interface byte if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte where the data is to be stored. |
|-------------|---|

Returns

BLT_TRUE if a byte was received, BLT_FALSE otherwise.

7.317.2.9 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.317.2.10 UsbReceivePipeBulkOUT()

```
void UsbReceivePipeBulkOUT (
    blt_int8u epnum )
```

Stores data that was received on the Bulk OUT pipe in the fifo.

Returns

none.

7.317.2.11 UsbTransmitByte()

```
static blt_bool UsbTransmitByte (
    blt_int8u data ) [static]
```

Transmits a communication interface byte.

Parameters

| | |
|-------------|--|
| <i>data</i> | Value of byte that is to be transmitted. |
|-------------|--|

Returns

BLT_TRUE if the byte was transmitted, BLT_FALSE otherwise.

Referenced by [UsbTransmitPacket\(\)](#).

7.317.2.12 UsbTransmitPacket()

```
void UsbTransmitPacket (   
    blt_int8u * data,  
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.317.2.13 UsbTransmitPipeBulkIN()

```
void UsbTransmitPipeBulkIN (   
    void )
```

Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.

Returns

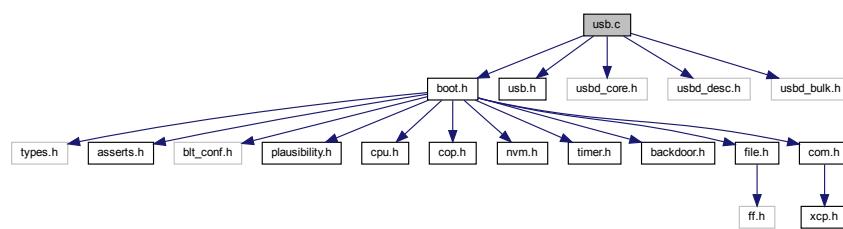
none.

7.318 usb.c File Reference

Bootloader USB communication interface source file.

```
#include "boot.h"
#include "usb.h"
#include "usbd_core.h"
#include "usbd_desc.h"
#include "usbd_bulk.h"

Include dependency graph for ARMCM7_STM32H7/usb.c:
```



Data Structures

- struct [tFifoCtrl](#)
Structure type for fifo control.
- struct [tFifoPipe](#)
Structure type for a fifo pipe.

Macros

- #define **FIFO_MAX_BUFFERS** (2)
Total number of fifo buffers.
- #define **FIFO_ERR_INVALID_HANDLE** (255)
Invalid value for a fifo buffer handle.
- #define **FIFO_PIPE_SIZE** (64)
Number of bytes that fit in the fifo pipe.

Functions

- static [blt_bool UsbReceiveByte](#) ([blt_int8u](#) *data)
Receives a communication interface byte if one is present.
- static [blt_bool UsbTransmitByte](#) ([blt_int8u](#) data)
Transmits a communication interface byte.
- static void [UsbFifoMgrInit](#) (void)
Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.
- static [blt_int8u UsbFifoMgrCreate](#) ([blt_int8u](#) *buffer, [blt_int8u](#) length)

- Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.*
- static `blt_bool UsbFifoMgrWrite (blt_int8u handle, blt_int8u data)`

Stores data in the fifo.
 - static `blt_bool UsbFifoMgrRead (blt_int8u handle, blt_int8u *data)`

Retrieves data from the fifo.
 - static `blt_int8u UsbFifoMgrScan (blt_int8u handle)`

Returns the number of data entries currently present in the fifo.
 - void `UsbInit (void)`

Initializes the USB communication interface.
 - void `UsbFree (void)`

Releases the USB communication interface.
 - void `UsbTransmitPacket (blt_int8u *data, blt_int8u len)`

Transmits a packet formatted for the communication interface.
 - `blt_bool UsbReceivePacket (blt_int8u *data, blt_int8u *len)`

Receives a communication interface packet if one is present.
 - void `UsbTransmitPipeBulkIN (void)`

Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.
 - void `UsbReceivePipeBulkOUT (blt_int8u epnum)`

Stores data that was received on the Bulk OUT pipe in the fifo.

Variables

- static `tFifoCtrl fifoCtrl [FIFO_MAX_BUFFERS]`

Local variable that holds the fifo control structures.
- static `tFifoCtrl * fifoCtrlFree`

Local pointer that points to the next free fifo control structure.
- static `tFifoPipe fifoPipeBulkIN`

Fifo pipe used for the bulk in endpoint.
- static `tFifoPipe fifoPipeBulkOUT`

Fifo pipe used for the bulk out endpoint.
- static `USBD_HandleTypeDef hUsbDeviceFS`

USB device handle.

7.318.1 Detailed Description

Bootloader USB communication interface source file.

7.318.2 Function Documentation

7.318.2.1 UsbFifoMgrCreate()

```
static blt_int8u UsbFifoMgrCreate (
    blt_int8u * buffer,
    blt_int8u length ) [static]
```

Places a data storage array under fifo management control. A handle for identifying the fifo in subsequent fifo management function calls is returned, if successful.

Parameters

| | |
|---------------|---|
| <i>buffer</i> | Pointer to the first element in the data storage fifo. |
| <i>length</i> | Maximum number of data elements that can be stored in the fifo. |

Returns

Fifo handle if successfull, or FIFO_ERR_INVALID_HANDLE.

Referenced by [UsbInit\(\)](#).

7.318.2.2 UsbFifoMgrInit()

```
static void UsbFifoMgrInit (
    void ) [static]
```

Initializes the fifo manager. Each controlled fifo is assigned a unique handle, which is the same as its index into fifoCtrl[]. Each controlled fifo holds a pointer to the next free fifo control. For the last fifo in fifoCtrl[] this one is set to a null-pointer as an out of fifo control indicator. Function should be called once before any of the other fifo management functions are called.

Returns

none.

Referenced by [UsbInit\(\)](#).

7.318.2.3 UsbFifoMgrRead()

```
static blt_bool UsbFifoMgrRead (
    blt_int8u handle,
    blt_int8u * data ) [static]
```

Retrieves data from the fifo.

Parameters

| | |
|---------------|---|
| <i>handle</i> | Identifies the fifo to read data from. |
| <i>data</i> | Pointer to where the read data is to be stored. |

Returns

BLT_TRUE if the data was successfully read from the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceiveByte\(\)](#), and [UsbTransmitPipeBulkIN\(\)](#).

7.318.2.4 UsbFifoMgrScan()

```
static blt_int8u UsbFifoMgrScan (
    blt_int8u handle ) [static]
```

Returns the number of data entries currently present in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo that is to be scanned. |
|---------------|--|

Returns

Number of data entries in the fifo if successful, otherwise 0.

Referenced by [UsbTransmitPipeBulkIN\(\)](#).

7.318.2.5 UsbFifoMgrWrite()

```
static blt_bool UsbFifoMgrWrite (
    blt_int8u handle,
    blt_int8u data ) [static]
```

Stores data in the fifo.

Parameters

| | |
|---------------|--|
| <i>handle</i> | Identifies the fifo to write data to. |
| <i>data</i> | Pointer to the data that is to be written to the fifo. |

Returns

BLT_TRUE if the data was successfully stored in the fifo, BLT_FALSE otherwise.

Referenced by [UsbReceivePipeBulkOUT\(\)](#), and [UsbTransmitByte\(\)](#).

7.318.2.6 UsbFree()

```
void UsbFree (
    void )
```

Releases the USB communication interface.

Returns

none.

7.318.2.7 UsbInit()

```
void UsbInit (
    void  )
```

Initializes the USB communication interface.

Returns

none.

7.318.2.8 UsbReceiveByte()

```
static blt_bool UsbReceiveByte (
    blt_int8u * data ) [static]
```

Receives a communication interface byte if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte where the data is to be stored. |
|-------------|---|

Returns

BLT_TRUE if a byte was received, BLT_FALSE otherwise.

7.318.2.9 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

7.318.2.10 UsbReceivePipeBulkOUT()

```
void UsbReceivePipeBulkOUT (
    blt_int8u eppnum )
```

Stores data that was received on the Bulk OUT pipe in the fifo.

Returns

none.

7.318.2.11 UsbTransmitByte()

```
static blt_bool UsbTransmitByte (
    blt_int8u data ) [static]
```

Transmits a communication interface byte.

Parameters

| | |
|-------------|--|
| <i>data</i> | Value of byte that is to be transmitted. |
|-------------|--|

Returns

BLT_TRUE if the byte was transmitted, BLT_FALSE otherwise.

Referenced by [UsbTransmitPacket\(\)](#).

7.318.2.12 UsbTransmitPacket()

```
void UsbTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

none.

7.318.2.13 UsbTransmitPipeBulkIN()

```
void UsbTransmitPipeBulkIN (
    void )
```

Checks if there is still data left to transmit and if so submits it for transmission with the USB endpoint.

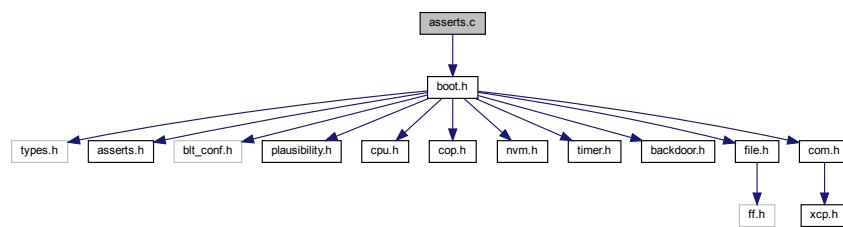
Returns

none.

7.319 asserts.c File Reference

Bootloader assertion module source file.

```
#include "boot.h"
Include dependency graph for asserts.c:
```



Functions

- void [AssertFailure](#) (`blt_char *file, blt_int32u line`)

Called when a runtime assertion failed. It stores information about where the assertion occurred and halts the software program.

7.319.1 Detailed Description

Bootloader assertion module source file.

7.319.2 Function Documentation

7.319.2.1 AssertFailure()

```
void AssertFailure (
    blt_char * file,
    blt_int32u line )
```

Called when a runtime assertion failed. It stores information about where the assertion occurred and halts the software program.

Parameters

| | |
|-------------|---|
| <i>file</i> | Name of the source file where the assertion occurred. |
| <i>line</i> | Linenumber in the source file where the assertion occurred. |

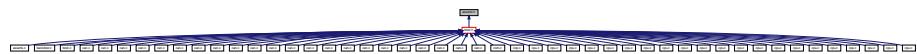
Returns

none

7.320 asserts.h File Reference

Bootloader assertion module header file.

This graph shows which files directly or indirectly include this file:



Macros

- #define **ASSERT_CT**(cond) enum { ASSERT_CONCAT(assert_error_on_line_, __LINE__) = 1/(!(cond)) }
Macro for assertions that can be performed at compile time.
- #define **ASSERT_RT**(cond)
Macro for assertions that can only be performed at run time.

Functions

- void [AssertFailure](#) (*blt_char* *file, *blt_int32u* line)
Called when a runtime assertion failed. It stores information about where the assertion occurred and halts the software program.

7.320.1 Detailed Description

Bootloader assertion module header file.

7.320.2 Function Documentation

7.320.2.1 AssertFailure()

```
void AssertFailure (
    blt_char * file,
    blt_int32u line )
```

Called when a runtime assertion failed. It stores information about where the assertion occurred and halts the software program.

Parameters

| | |
|-------------|---|
| <i>file</i> | Name of the source file where the assertion occurred. |
| <i>line</i> | Linenumber in the source file where the assertion occurred. |

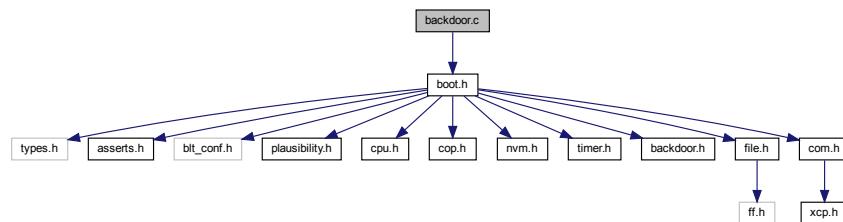
Returns

none

7.321 backdoor.c File Reference

Bootloader backdoor entry source file.

```
#include "boot.h"
Include dependency graph for backdoor.c:
```



Functions

- void [BackDoorInit](#) (void)
Initializes the backdoor entry option.
- void [BackDoorCheck](#) (void)
The default backdoor entry feature keeps the bootloader active for a predetermined time after reset, allowing the host application to establish a connection and start a programming sequence. This function controls the opening/closing of the backdoor.

7.321.1 Detailed Description

Bootloader backdoor entry source file.

7.321.2 Function Documentation

7.321.2.1 BackDoorCheck()

```
void BackDoorCheck (
    void )
```

The default backdoor entry feature keeps the bootloader active for a predetermined time after reset, allowing the host application to establish a connection and start a programming sequence. This function controls the opening/closing of the backdoor.

Returns

none

Referenced by [BootTask\(\)](#).

7.321.2.2 BackDoorInit()

```
void BackDoorInit (
    void )
```

Initializes the backdoor entry option.

Returns

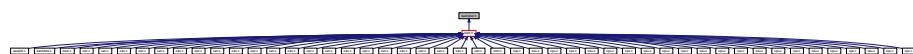
none

Referenced by [BootInit\(\)](#).

7.322 backdoor.h File Reference

Bootloader backdoor entry header file.

This graph shows which files directly or indirectly include this file:



Functions

- void [BackDoorInit](#) (void)

Initializes the backdoor entry option.

- void [BackDoorCheck](#) (void)

The default backdoor entry feature keeps the bootloader active for a predetermined time after reset, allowing the host application to establish a connection and start a programming sequence. This function controls the opening/closing of the backdoor.

7.322.1 Detailed Description

Bootloader backdoor entry header file.

7.322.2 Function Documentation

7.322.2.1 BackDoorCheck()

```
void BackDoorCheck (
    void )
```

The default backdoor entry feature keeps the bootloader active for a predetermined time after reset, allowing the host application to establish a connection and start a programming sequence. This function controls the opening/closing of the backdoor.

Returns

none

Referenced by [BootTask\(\)](#).

7.322.2.2 BackDoorInit()

```
void BackDoorInit (
    void )
```

Initializes the backdoor entry option.

Returns

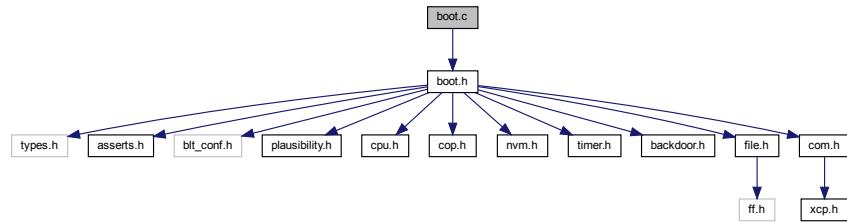
none

Referenced by [BootInit\(\)](#).

7.323 boot.c File Reference

Bootloader core module source file.

```
#include "boot.h"
Include dependency graph for boot.c:
```



Functions

- void [BootInit](#) (void)
Initializes the bootloader core.
- void [BootTask](#) (void)
Task function of the bootloader core that drives the program.

7.323.1 Detailed Description

Bootloader core module source file.

7.323.2 Function Documentation

7.323.2.1 BootInit()

```
void BootInit (
    void )
```

Initializes the bootloader core.

Returns

none

7.323.2.2 BootTask()

```
void BootTask (
    void )
```

Task function of the bootloader core that drives the program.

Returns

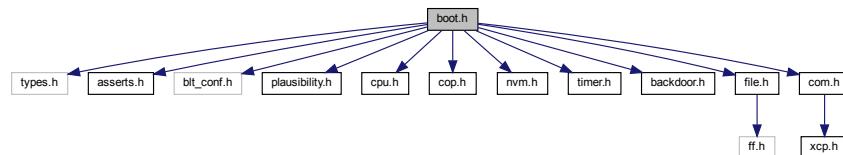
none

7.324 boot.h File Reference

Bootloader core module header file.

```
#include "types.h"
#include "asserts.h"
#include "blt_conf.h"
#include "plausibility.h"
#include "cpu.h"
#include "cop.h"
#include "nvm.h"
#include "timer.h"
#include "backdoor.h"
#include "file.h"
#include "com.h"
```

Include dependency graph for boot.h:



Macros

- `#define BOOT_VERSION_CORE_MAIN (1u)`
Main version number of the bootloader core.
- `#define BOOT_VERSION_CORE_MINOR (19u)`
Minor version number of the bootloader core.
- `#define BOOT_VERSION_CORE_PATCH (1u)`
Patch number of the bootloader core.

Functions

- `void BootInit (void)`
Initializes the bootloader core.
- `void BootTask (void)`
Task function of the bootloader core that drives the program.

7.324.1 Detailed Description

Bootloader core module header file.

7.324.2 Function Documentation

7.324.2.1 BootInit()

```
void BootInit (
    void )
```

Initializes the bootloader core.

Returns

none

7.324.2.2 BootTask()

```
void BootTask (
    void )
```

Task function of the bootloader core that drives the program.

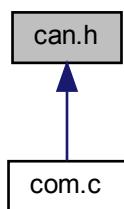
Returns

none

7.325 can.h File Reference

Bootloader CAN communication interface header file.

This graph shows which files directly or indirectly include this file:



Functions

- void [CanInit](#) (void)
Initializes the CAN controller and synchronizes it to the CAN bus.
- void [CanTransmitPacket](#) ([blt_int8u](#) *data, [blt_int8u](#) len)
Transmits a packet formatted for the communication interface.
- [blt_bool](#) [CanReceivePacket](#) ([blt_int8u](#) *data, [blt_int8u](#) *len)
Receives a communication interface packet if one is present.

7.325.1 Detailed Description

Bootloader CAN communication interface header file.

7.325.2 Function Documentation

7.325.2.1 CanInit()

```
void CanInit (
    void )
```

Initializes the CAN controller and synchronizes it to the CAN bus.

Returns

none.

Referenced by [ComInit\(\)](#).

7.325.2.2 CanReceivePacket()

```
blt_bool CanReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

`BLT_TRUE` is a packet was received, `BLT_FALSE` otherwise.

Referenced by [ComTask\(\)](#).

7.325.2.3 CanTransmitPacket()

```
void CanTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------------|--|
| <code>data</code> | Pointer to byte array with data that it to be transmitted. |
| <code>len</code> | Number of bytes that are to be transmitted. |

Returns

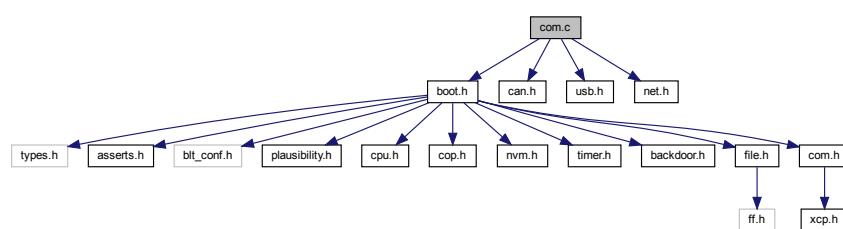
none.

Referenced by [ComTransmitPacket\(\)](#).

7.326 com.c File Reference

Bootloader communication interface source file.

```
#include "boot.h"
#include "can.h"
#include "usb.h"
#include "net.h"
Include dependency graph for com.c:
```



Functions

- void **ComInit** (void)
Initializes the communication module including the hardware needed for the communication.
- void **ComTask** (void)
Updates the communication module by checking if new data was received and submitting the request to process newly received data.
- void **ComFree** (void)
Releases the communication module.
- void **ComTransmitPacket** (**blt_int8u** *data, **blt_int16u** len)
Transmits the packet using the xcp transport layer.
- **blt_int16u** **ComGetActiveInterfaceMaxRxLen** (void)
Obtains the maximum number of bytes that can be received on the specified communication interface.
- **blt_int16u** **ComGetActiveInterfaceMaxTxLen** (void)
Obtains the maximum number of bytes that can be transmitted on the specified communication interface.
- **blt_bool** **ComIsConnected** (void)
This function obtains the XCP connection state.

Variables

- static **tComInterfaceId** **comActiveInterface** = **COM_IF_OTHER**
Holds the communication interface of the currently active interface.

7.326.1 Detailed Description

Bootloader communication interface source file.

7.326.2 Function Documentation

7.326.2.1 ComFree()

```
void ComFree (
    void )
```

Releases the communication module.

Returns

none

7.326.2.2 ComGetActiveInterfaceMaxRxLen()

```
blt_int16u ComGetActiveInterfaceMaxRxLen (
    void )
```

Obtains the maximum number of bytes that can be received on the specified communication interface.

Returns

Maximum number of bytes that can be received.

7.326.2.3 ComGetActiveInterfaceMaxTxLen()

```
blt_int16u ComGetActiveInterfaceMaxTxLen (
    void )
```

Obtains the maximum number of bytes that can be transmitted on the specified communication interface.

Returns

Maximum number of bytes that can be received.

7.326.2.4 ComInit()

```
void ComInit (
    void )
```

Initializes the communication module including the hardware needed for the communication.

Returns

none

Referenced by [BootInit\(\)](#).

7.326.2.5 ComIsConnected()

```
blt_bool ComIsConnected (
    void )
```

This function obtains the XCP connection state.

Returns

BLT_TRUE when an XCP connection is established, BLT_FALSE otherwise.

Referenced by [BackDoorCheck\(\)](#), and [FileHandleFirmwareUpdateRequest\(\)](#).

7.326.2.6 ComTask()

```
void ComTask (
    void  )
```

Updates the communication module by checking if new data was received and submitting the request to process newly received data.

Returns

none

Referenced by [BootTask\(\)](#).

7.326.2.7 ComTransmitPacket()

```
void ComTransmitPacket (
    blt_int8u * data,
    blt_int16u len )
```

Transmits the packet using the xcp transport layer.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to the byte buffer with packet data. |
| <i>len</i> | Number of data bytes that need to be transmitted. |

Returns

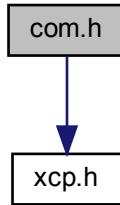
none

Referenced by [XcpTransmitPacket\(\)](#).

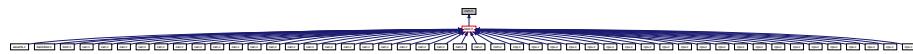
7.327 com.h File Reference

Bootloader communication interface header file.

```
#include "xcp.h"
Include dependency graph for com.h:
```



This graph shows which files directly or indirectly include this file:



Macros

- `#define BOOT_COM_RX_MAX_DATA (1)`
Defines the maximum number of bytes for transport layer reception depending on the activates interface(s).
- `#define BOOT_COM_RX_MAX_DATA (BOOT_COM_CAN_RX_MAX_DATA)`
Defines the maximum number of bytes for transport layer reception depending on the activates interface(s).
- `#define BOOT_COM_RX_MAX_DATA (BOOT_COM_NET_RX_MAX_DATA)`
Defines the maximum number of bytes for transport layer reception depending on the activates interface(s).
- `#define BOOT_COM_TX_MAX_DATA (1)`
Defines the maximum number of bytes for transport layer transmission depending on the activates interface(s).
- `#define BOOT_COM_TX_MAX_DATA (BOOT_COM_CAN_TX_MAX_DATA)`
Defines the maximum number of bytes for transport layer transmission depending on the activates interface(s).
- `#define BOOT_COM_TX_MAX_DATA (BOOT_COM_USB_TX_MAX_DATA)`
Defines the maximum number of bytes for transport layer transmission depending on the activates interface(s).
- `#define BOOT_COM_TX_MAX_DATA (BOOT_COM_NET_TX_MAX_DATA)`
Defines the maximum number of bytes for transport layer transmission depending on the activates interface(s).

Enumerations

- `enum tComInterfaceId {
 COM_IF_RS232 , COM_IF_MBRTU , COM_IF_CAN , COM_IF_USB ,
 COM_IF_NET , COM_IF_OTHER }`

Enumeration for the different communication interfaces.

Functions

- void [ComInit](#) (void)
Initializes the communication module including the hardware needed for the communication.
- void [ComTask](#) (void)
Updates the communication module by checking if new data was received and submitting the request to process newly received data.
- void [ComFree](#) (void)
Releases the communication module.
- [blt_int16u ComGetActiveInterfaceMaxRxLen](#) (void)
Obtains the maximum number of bytes that can be received on the specified communication interface.
- [blt_int16u ComGetActiveInterfaceMaxTxLen](#) (void)
Obtains the maximum number of bytes that can be transmitted on the specified communication interface.
- void [ComTransmitPacket](#) ([blt_int8u](#) *data, [blt_int16u](#) len)
Transmits the packet using the xcp transport layer.
- [blt_bool ComIsConnected](#) (void)
This function obtains the XCP connection state.

7.327.1 Detailed Description

Bootloader communication interface header file.

7.327.2 Enumeration Type Documentation

7.327.2.1 tComInterfaceId

```
enum tComInterfaceId
```

Enumeration for the different communication interfaces.

Enumerator

| | |
|------------------------------|----------------------|
| COM_IF_RS232 | RS232 interface |
| COM_IF_MBRTU | Modbus RTU interface |
| COM_IF_CAN | CAN interface |
| COM_IF_USB | USB interface |
| COM_IF_NET | NET interface |
| COM_IF_OTHER | Other interface |

7.327.3 Function Documentation

7.327.3.1 ComFree()

```
void ComFree (
    void )
```

Releases the communication module.

Returns

none

7.327.3.2 ComGetActiveInterfaceMaxRxLen()

```
blt_int16u ComGetActiveInterfaceMaxRxLen (
    void )
```

Obtains the maximum number of bytes that can be received on the specified communication interface.

Returns

Maximum number of bytes that can be received.

7.327.3.3 ComGetActiveInterfaceMaxTxLen()

```
blt_int16u ComGetActiveInterfaceMaxTxLen (
    void )
```

Obtains the maximum number of bytes that can be transmitted on the specified communication interface.

Returns

Maximum number of bytes that can be received.

7.327.3.4 ComInit()

```
void ComInit (
    void )
```

Initializes the communication module including the hardware needed for the communication.

Returns

none

Referenced by [BootInit\(\)](#).

7.327.3.5 ComIsConnected()

```
blt_bool ComIsConnected (
    void )
```

This function obtains the XCP connection state.

Returns

BLT_TRUE when an XCP connection is established, BLT_FALSE otherwise.

Referenced by [BackDoorCheck\(\)](#), and [FileHandleFirmwareUpdateRequest\(\)](#).

7.327.3.6 ComTask()

```
void ComTask (
    void )
```

Updates the communication module by checking if new data was received and submitting the request to process newly received data.

Returns

none

Referenced by [BootTask\(\)](#).

7.327.3.7 ComTransmitPacket()

```
void ComTransmitPacket (
    blt_int8u * data,
    blt_int16u len )
```

Transmits the packet using the xcp transport layer.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to the byte buffer with packet data. |
| <i>len</i> | Number of data bytes that need to be transmitted. |

Returns

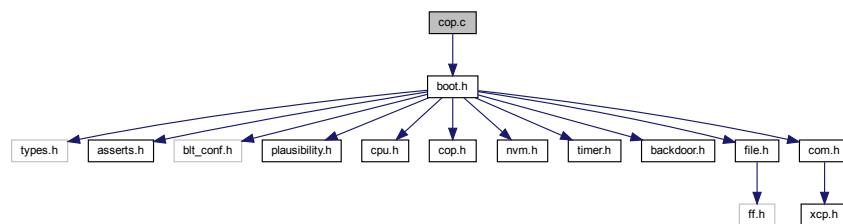
none

Referenced by [XcpTransmitPacket\(\)](#).

7.328 cop.c File Reference

Bootloader watchdog module source file.

```
#include "boot.h"
Include dependency graph for cop.c:
```



Functions

- void [CopInit](#) (void)
Watchdog initialization function.
- void [CopService](#) (void)
Watchdog service function to prevent the watchdog from timing out.

7.328.1 Detailed Description

Bootloader watchdog module source file.

7.328.2 Function Documentation

7.328.2.1 CopInit()

```
void CopInit (
    void )
```

Watchdog initialization function.

Returns

none

Referenced by [BootInit\(\)](#).

7.328.2.2 CopService()

```
void CopService (
    void )
```

Watchdog service function to prevent the watchdog from timing out.

Returns

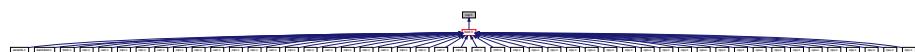
none

Referenced by [AssertFailure\(\)](#), [BootTask\(\)](#), [CanDisabledModeEnter\(\)](#), [CanDisabledModeExit\(\)](#), [CanFreezeModeEnter\(\)](#), [CanFreezeModeExit\(\)](#), [CanGetSpeedConfig\(\)](#), [CanInit\(\)](#), [CanTransmitPacket\(\)](#), [CpuMemCopy\(\)](#), [CpuMemSet\(\)](#), [FlashAddToBlock\(\)](#), [FlashEmptyCheckSector\(\)](#), [FlashErase\(\)](#), [FlashEraseSectors\(\)](#), [FlashGetSector\(\)](#), [FlashGetSectorBaseAddr\(\)](#), [FlashGetSectorIdx\(\)](#), [FlashGetSectorSize\(\)](#), [FlashWriteBlock\(\)](#), [UsbTransmitPacket\(\)](#), and [XcpComputeChecksum\(\)](#).

7.329 cop.h File Reference

Bootloader watchdog module header file.

This graph shows which files directly or indirectly include this file:



Functions

- void [CopInit](#) (void)
Watchdog initialization function.
- void [CopService](#) (void)
Watchdog service function to prevent the watchdog from timing out.

7.329.1 Detailed Description

Bootloader watchdog module header file.

7.329.2 Function Documentation

7.329.2.1 CopInit()

```
void CopInit (
    void )
```

Watchdog initialization function.

Returns

none

Referenced by [BootInit\(\)](#).

7.329.2.2 CopService()

```
void CopService (
    void )
```

Watchdog service function to prevent the watchdog from timing out.

Returns

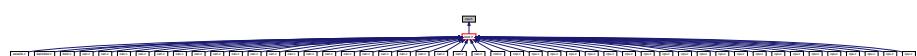
none

Referenced by [AssertFailure\(\)](#), [BootTask\(\)](#), [CanDisabledModeEnter\(\)](#), [CanDisabledModeExit\(\)](#), [CanFreezeModeEnter\(\)](#), [CanFreezeModeExit\(\)](#), [CanGetSpeedConfig\(\)](#), [CanInit\(\)](#), [CanTransmitPacket\(\)](#), [CpuMemcpy\(\)](#), [CpuMemSet\(\)](#), [FlashAddToBlock\(\)](#), [FlashEmptyCheckSector\(\)](#), [FlashErase\(\)](#), [FlashEraseSectors\(\)](#), [FlashGetSector\(\)](#), [FlashGetSectorBaseAddr\(\)](#), [FlashGetSectorIdx\(\)](#), [FlashGetSectorSize\(\)](#), [FlashWriteBlock\(\)](#), [UsbTransmitPacket\(\)](#), and [XcpComputeChecksum\(\)](#).

7.330 cpu.h File Reference

Bootloader cpu module header file.

This graph shows which files directly or indirectly include this file:



Functions

- void [CpuInit](#) (void)
Initializes the CPU module.
- void [CpuStartUserProgram](#) (void)
Starts the user program, if one is present. In this case this function does not return.
- void [CpuMemcpy](#) ([blt_addr](#) dest, [blt_addr](#) src, [blt_int16u](#) len)
Copies data from the source to the destination address.
- void [CpuMemSet](#) ([blt_addr](#) dest, [blt_int8u](#) value, [blt_int16u](#) len)
Sets the bytes at the destination address to the specified value.
- void [CpuIrqDisable](#) (void)
Disable global interrupts.
- void [CpuIrqEnable](#) (void)
Enable global interrupts.

7.330.1 Detailed Description

Bootloader cpu module header file.

7.330.2 Function Documentation

7.330.2.1 [CpuInit\(\)](#)

```
void CpuInit (
    void )
```

Initializes the CPU module.

Returns

none.

Referenced by [BootInit\(\)](#).

7.330.2.2 [CpuIrqDisable\(\)](#)

```
void CpuIrqDisable (
    void )
```

Disable global interrupts.

Returns

none.

Referenced by [CpuInit\(\)](#).

7.330.2.3 CpuIrqEnable()

```
void CpuIrqEnable (
    void )
```

Enable global interrupts.

Returns

none.

7.330.2.4 CpuMemcpy()

```
void CpuMemcpy (
    blt_addr dest,
    blt_addr src,
    blt_int16u len )
```

Copies data from the source to the destination address.

Parameters

| | |
|-------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>src</i> | Source address of the data. |
| <i>len</i> | length of the data in bytes. |

Returns

none.

Referenced by [FlashInitBlock\(\)](#), [XcpCmdGetSeed\(\)](#), [XcpCmdShortUpload\(\)](#), [XcpCmdUnlock\(\)](#), and [XcpCmdUpload\(\)](#).

7.330.2.5 CpuMemSet()

```
void CpuMemSet (
    blt_addr dest,
    blt_int8u value,
    blt_int16u len )
```

Sets the bytes at the destination address to the specified value.

Parameters

| | |
|--------------|-----------------------------------|
| <i>dest</i> | Destination address for the data. |
| <i>value</i> | Value to write. |
| <i>len</i> | Number of bytes to write. |

Returns

none.

Referenced by [XcpCmdShortUpload\(\)](#), and [XcpCmdUpload\(\)](#).

7.330.2.6 CpuStartUserProgram()

```
void CpuStartUserProgram (
    void )
```

Starts the user program, if one is present. In this case this function does not return.

Returns

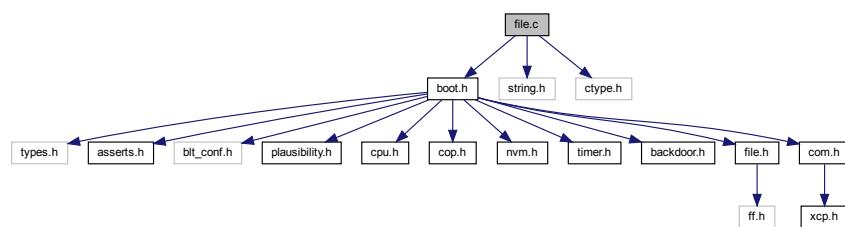
none.

Referenced by [BackDoorCheck\(\)](#), and [XcpCmdProgramReset\(\)](#).

7.331 file.c File Reference

Bootloader file system interface source file.

```
#include "boot.h"
#include <string.h>
#include <ctype.h>
Include dependency graph for file.c:
```

**Data Structures**

- struct [tFileEraseInfo](#)
Structure type with information for the memory erase opeartion.
- struct [tFatFsObjects](#)
Structure type for grouping FATFS related objects used by this module.

Enumerations

- enum **tFirmwareUpdateState** { **FIRMWARE_UPDATE_STATE_IDLE** , **FIRMWARE_UPDATE_STATE_STARTING** , **FIRMWARE_UPDATE_STATE_ERASING** , **FIRMWARE_UPDATE_STATE_PROGRAMMING** }
- Enumeration for the different internal module states.*

Functions

- static **blt_char FileLibByteNibbleToChar (blt_int8u nibble)**
Helper function to convert a 4-bit value to a character that represents its value in hexadecimal format. Example: FileLibByteNibbleToChar(11) --> returns 'B'.
- static **blt_char * FileLibByteToHexString (blt_int8u byte_val, blt_char *destination)**
Helper function to convert a byte value to a string representing the value in hexadecimal format. Example: FileLibByteToHexString(28, strBuffer) --> returns "1C".
- static **blt_char * FileLibLongToIntString (blt_int32u long_val, blt_char *destination)**
Helper function to convert a 32-bit unsigned number to a string that represents its decimal value. Example: FileLibLongToIntString(1234, strBuffer) --> returns "1234".
- static **blt_int8u FileLibHexStringToByte (const blt_char *hexstring)**
Helper function to convert a sequence of 2 characters that represent a hexadecimal value to the actual byte value. Example: FileLibHexStringToByte("2f") --> returns 47.
- void **FileInit (void)**
Initializes the file system interface module. The initial firmware update state is set to idle and the file system is mounted as logical disk 0.
- **blt_bool FileIsIdle (void)**
This function checks if a firmware update through the locally attached storage is in progress or not (idle).
- **blt_bool FileHandleFirmwareUpdateRequest (void)**
This function checks if a firmware update through the locally attached storage is requested to be started and if so processes this request by transitioning from the IDLE to the STARTING state.
- void **FileTask (void)**
File system task function for managing the firmware updates from locally attached storage.
- **tSrecLineType FileSrecGetLineType (const blt_char *line)**
Inspects a line from a Motorola S-Record file to determine its type.
- **blt_bool FileSrecVerifyChecksum (const blt_char *line)**
Inspects an S1, S2 or S3 line from a Motorola S-Record file to determine if the checksum at the end is correct.
- **blt_int16s FileSrecParseLine (const blt_char *line, blt_addr *address, blt_int8u *data)**
Parses a line from a Motorola S-Record file and looks for S1, S2 or S3 lines with data. Note that if a null pointer is passed as the data parameter, then no data is extracted from the line.

Variables

- static **tFirmwareUpdateState firmwareUpdateState**
Local variable that holds the internal module state.
- static **tFatFsObjects fatFsObjects**
Local variable for the used FATFS objects in this module.
- static **tSrecLineParseObject lineParseObject**
Local variable for storing S-record line parsing results.
- static **tFileEraseInfo eraseInfo**
Local variable for storing information regarding the memory erase operation.
- static **blt_char loggingStr [64]**
Local character buffer for storing the string with log information.

7.331.1 Detailed Description

Bootloader file system interface source file.

7.331.2 Enumeration Type Documentation

7.331.2.1 tFirmwareUpdateState

enum [tFirmwareUpdateState](#)

Enumeration for the different internal module states.

Enumerator

| | |
|-----------------------------------|-------------------|
| FIRMWARE_UPDATE_STATE_IDLE | idle state |
| FIRMWARE_UPDATE_STATE_STARTING | starting state |
| FIRMWARE_UPDATE_STATE_ERASING | erasing state |
| FIRMWARE_UPDATE_STATE_PROGRAMMING | programming state |

7.331.3 Function Documentation

7.331.3.1 FileHandleFirmwareUpdateRequest()

```
blt_bool FileHandleFirmwareUpdateRequest (
    void )
```

This function checks if a firmware update through the locally attached storage is requested to be started and if so processes this request by transitioning from the IDLE to the STARTING state.

Returns

BLT_TRUE when a firmware update is requested, BLT_FALSE otherwise.

Referenced by [BackDoorCheck\(\)](#).

7.331.3.2 FileInit()

```
void FileInit (
    void  )
```

Initializes the file system interface module. The initial firmware update state is set to idle and the file system is mounted as logical disk 0.

Returns

none

Referenced by [BootInit\(\)](#).

7.331.3.3 FileIsIdle()

```
blt_bool FileIsIdle (
    void  )
```

This function checks if a firmware update through the locally attached storage is in progress or not (idle).

Returns

BLT_TRUE when in idle state, BLT_FALSE otherwise.

Referenced by [BackDoorCheck\(\)](#), and [XcpCmdConnect\(\)](#).

7.331.3.4 FileLibByteNibbleToChar()

```
static blt_char FileLibByteNibbleToChar (
    blt_int8u nibble ) [static]
```

Helper function to convert a 4-bit value to a character that represents its value in hexadecimal format. Example: FileLibByteNibbleToChar(11) --> returns 'B'.

Parameters

| | |
|---------------|-------------------------|
| <i>nibble</i> | 4-bit value to convert. |
|---------------|-------------------------|

Returns

The resulting byte value.

Referenced by [FileLibByteToHexString\(\)](#).

7.331.3.5 FileLibByteToHexString()

```
static blt_char * FileLibByteToHexString (
    blt_int8u byte_val,
    blt_char * destination ) [static]
```

Helper function to convert a byte value to a string representing the value in hexadecimal format. Example: FileLibByteToHexString(28, strBuffer) --> returns "1C".

Parameters

| | |
|--------------------|--|
| <i>byte_val</i> | 8-bit value to convert. |
| <i>destination</i> | Pointer to character buffer for storing the results. |

Returns

The resulting string.

7.331.3.6 FileLibHexStringToByte()

```
static blt_int8u FileLibHexStringToByte (
    const blt_char * hexstring ) [static]
```

Helper function to convert a sequence of 2 characters that represent a hexadecimal value to the actual byte value. Example: FileLibHexStringToByte("2f") --> returns 47.

Parameters

| | |
|------------------|---|
| <i>hexstring</i> | String beginning with 2 characters that represent a hexa-decimal value. |
|------------------|---|

Returns

The resulting byte value.

Referenced by [FileSrecParseLine\(\)](#), and [FileSrecVerifyChecksum\(\)](#).

7.331.3.7 FileLibLongToIntString()

```
static blt_char * FileLibLongToIntString (
    blt_int32u long_val,
    blt_char * destination ) [static]
```

Helper function to convert a 32-bit unsigned number to a string that represents its decimal value. Example: FileLibLongToIntString(1234, strBuffer) --> returns "1234".

Parameters

| | |
|--------------------|--|
| <i>long_val</i> | 32-bit value to convert. |
| <i>destination</i> | Pointer to character buffer for storing the results. |

Returns

The resulting string.

7.331.3.8 FileSrecGetLineType()

```
tSrecLineType FileSrecGetLineType (
    const blt_char * line )
```

Inspects a line from a Motorola S-Record file to determine its type.

Parameters

| | |
|-------------|---------------------------|
| <i>line</i> | A line from the S-Record. |
|-------------|---------------------------|

Returns

the S-Record line type.

Referenced by [FileSrecParseLine\(\)](#).

7.331.3.9 FileSrecParseLine()

```
blt_int16s FileSrecParseLine (
    const blt_char * line,
    blt_addr * address,
    blt_int8u * data )
```

Parses a line from a Motorola S-Record file and looks for S1, S2 or S3 lines with data. Note that if a null pointer is passed as the data parameter, then no data is extracted from the line.

Parameters

| | |
|----------------|---|
| <i>line</i> | A line from the S-Record. |
| <i>address</i> | Address found in the S-Record data line. |
| <i>data</i> | Byte array where the data bytes from the S-Record data line are stored. |

Returns

The number of data bytes found on the S-record data line, 0 in case the line is not an S1, S2 or S3 line or ERROR_SREC_INVALID_CHECKSUM in case the checksum validation failed.

7.331.3.10 FileSrecVerifyChecksum()

```
blt_bool FileSrecVerifyChecksum (
    const blt_char * line )
```

Inspects an S1, S2 or S3 line from a Motorola S-Record file to determine if the checksum at the end is correct.

Parameters

| | |
|-------------|---|
| <i>line</i> | An S1, S2 or S3 line from the S-Record. |
|-------------|---|

Returns

BLT_TRUE if the checksum is correct, BLT_FALSE otherwise.

Referenced by [FileSrecParseLine\(\)](#).

7.331.3.11 FileTask()

```
void FileTask (
    void )
```

File system task function for managing the firmware updates from locally attached storage.

Returns

none.

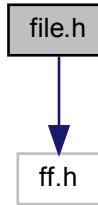
Referenced by [BootTask\(\)](#).

7.332 file.h File Reference

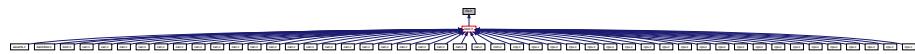
Bootloader file system interface header file.

```
#include "ff.h"
```

Include dependency graph for file.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct [tSrecLineParseObject](#)
Structure type for grouping the parsing results of an S-record line.

Macros

- #define **FILE_ERROR_CANNOT_OPEN_FIRMWARE_FILE** (1)
Error code for not being able to open the firmware file.
- #define **FILE_ERROR_CANNOT_READ_FROM_FILE** (2)
Error code for not being able to read from the firmware file.
- #define **FILE_ERROR_INVALID_CHECKSUM_IN_FILE** (3)
Error code because an incorrect checksum was found in the firmware file.
- #define **FILE_ERRORREWINDINGFILEREAD_POINTER** (4)
Error code because the file pointers read pointer could not be rewinded.
- #define **FILE_ERROR_CANNOT_ERASE_MEMORY** (5)
Error code because an error occurred during the memory erase operation.
- #define **FILE_ERROR_CANNOT_PROGRAM_MEMORY** (6)
Error code because an error occurred during the memory write operation.
- #define **FILE_ERROR_CANNOT_WRITE_CHECKSUM** (7)
Error code because the program's checksum could not be written to memory.
- #define **MAX_CHARS_PER_LINE** (256)
Maximum number of characters that can be on a line in the firmware file.
- #define **MAX_DATA_BYTES_PER_LINE** ([MAX_CHARS_PER_LINE](#)/2)
Maximum number of data bytes that can be on a line in the firmware file (S-record).
- #define **ERROR_SREC_INVALID_CHECKSUM** (-1)
Return code in case an invalid checksum was detected on an S-record line.

Enumerations

- enum `tSrecLineType` { `LINE_TYPE_S1` , `LINE_TYPE_S2` , `LINE_TYPE_S3` , `LINE_TYPE_UNSUPPORTED` }

Enumeration for the different S-record line types.

Functions

- void `FileInit` (void)
Initializes the file system interface module. The initial firmware update state is set to idle and the file system is mounted as logical disk 0.
- void `FileTask` (void)
File system task function for managing the firmware updates from locally attached storage.
- `blt_bool FileIsIdle` (void)
This function checks if a firmware update through the locally attached storage is in progress or not (idle).
- `blt_bool FileHandleFirmwareUpdateRequest` (void)
This function checks if a firmware update through the locally attached storage is requested to be started and if so processes this request by transitioning from the IDLE to the STARTING state.
- `tSrecLineType FileSrecGetLineType` (const `blt_char` *line)
Inspects a line from a Motorola S-Record file to determine its type.
- `blt_bool FileSrecVerifyChecksum` (const `blt_char` *line)
Inspects an S1, S2 or S3 line from a Motorola S-Record file to determine if the checksum at the end is correct.
- `blt_int16s FileSrecParseLine` (const `blt_char` *line, `blt_addr` *address, `blt_int8u` *data)
Parses a line from a Motorola S-Record file and looks for S1, S2 or S3 lines with data. Note that if a null pointer is passed as the data parameter, then no data is extracted from the line.

7.332.1 Detailed Description

Bootloader file system interface header file.

7.332.2 Enumeration Type Documentation

7.332.2.1 `tSrecLineType`

```
enum tSrecLineType
```

Enumeration for the different S-record line types.

Enumerator

| | |
|------------------------------------|---------------------|
| <code>LINE_TYPE_S1</code> | 16-bit address line |
| <code>LINE_TYPE_S2</code> | 24-bit address line |
| <code>LINE_TYPE_S3</code> | 32-bit address line |
| <code>LINE_TYPE_UNSUPPORTED</code> | unsupported line |

7.332.3 Function Documentation

7.332.3.1 FileHandleFirmwareUpdateRequest()

```
blt_bool FileHandleFirmwareUpdateRequest (
    void )
```

This function checks if a firmware update through the locally attached storage is requested to be started and if so processes this request by transitioning from the IDLE to the STARTING state.

Returns

BLT_TRUE when a firmware update is requested, BLT_FALSE otherwise.

Referenced by [BackDoorCheck\(\)](#).

7.332.3.2 FileInit()

```
void FileInit (
    void )
```

Initializes the file system interface module. The initial firmware update state is set to idle and the file system is mounted as logical disk 0.

Returns

none

Referenced by [BootInit\(\)](#).

7.332.3.3 FileIsIdle()

```
blt_bool FileIsIdle (
    void )
```

This function checks if a firmware update through the locally attached storage is in progress or not (idle).

Returns

BLT_TRUE when in idle state, BLT_FALSE otherwise.

Referenced by [BackDoorCheck\(\)](#), and [XcpCmdConnect\(\)](#).

7.332.3.4 FileSrecGetLineType()

```
tSrecLineType FileSrecGetLineType (
    const blt_char * line )
```

Inspects a line from a Motorola S-Record file to determine its type.

Parameters

| | |
|-------------|---------------------------|
| <i>line</i> | A line from the S-Record. |
|-------------|---------------------------|

Returns

the S-Record line type.

Referenced by [FileSrecParseLine\(\)](#).

7.332.3.5 FileSrecParseLine()

```
blt_int16s FileSrecParseLine (
    const blt_char * line,
    blt_addr * address,
    blt_int8u * data )
```

Parses a line from a Motorola S-Record file and looks for S1, S2 or S3 lines with data. Note that if a null pointer is passed as the data parameter, then no data is extracted from the line.

Parameters

| | |
|----------------|---|
| <i>line</i> | A line from the S-Record. |
| <i>address</i> | Address found in the S-Record data line. |
| <i>data</i> | Byte array where the data bytes from the S-Record data line are stored. |

Returns

The number of data bytes found on the S-record data line, 0 in case the line is not an S1, S2 or S3 line or ERROR_SREC_INVALID_CHECKSUM in case the checksum validation failed.

7.332.3.6 FileSrecVerifyChecksum()

```
blt_bool FileSrecVerifyChecksum (
    const blt_char * line )
```

Inspects an S1, S2 or S3 line from a Motorola S-Record file to determine if the checksum at the end is correct.

Parameters

| | |
|-------------|---|
| <i>line</i> | An S1, S2 or S3 line from the S-Record. |
|-------------|---|

Returns

BLT_TRUE if the checksum is correct, BLT_FALSE otherwise.

Referenced by [FileSrecParseLine\(\)](#).

7.332.3.7 FileTask()

```
void FileTask (
    void )
```

File system task function for managing the firmware updates from locally attached storage.

Returns

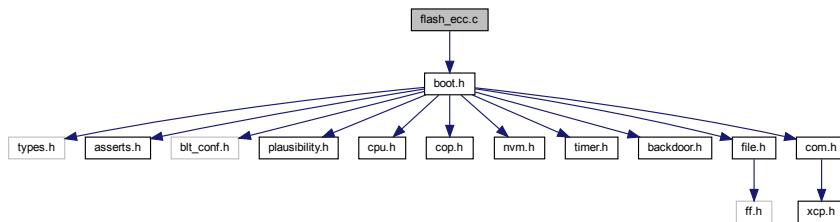
none.

Referenced by [BootTask\(\)](#).

7.333 flash_ecc.c File Reference

Bootloader flash driver source file for HCS12 derivatives with error correction code in flash memory, such as the HCS12Pxx. This flash memory uses a different addressing scheme than other HCS12 derivatives.

```
#include "boot.h"
Include dependency graph for flash_ecc.c:
```

**Data Structures**

- struct [tFlashSector](#)
Flash sector descriptor type.
- struct [tFlashBlockInfo](#)
Structure type for grouping flash block information.
- struct [tFlashRegs](#)
Structure type for the flash control registers.
- struct [tFlashPrescalerSysclockMapping](#)
Mapping table for finding the correct flash clock divider prescaler.

Macros

- #define **FLASH_INVALID_SECTOR_IDX** (0xff)
Value for an invalid flash sector.
- #define **FLASH_INVALID_ADDRESS** (0xffffffff)
Value for an invalid flash address.
- #define **FLASH_WRITE_BLOCK_SIZE** (512)
Standard size of a flash block for writing.
- #define **FLASH_TOTAL_SECTORS** (sizeof(**flashLayout**)/sizeof(**flashLayout[0]**))
*Total numbers of sectors in array **flashLayout**[].*
- #define **BOOT_FLASH_VECTOR_TABLE_CS_OFFSET** (0x82)
*Offset into the user program's vector table where the checksum is located. Note that the value can be overridden in **blt_conf.h**, because the size of the vector table could vary. When changing this value, don't forget to update the location of the checksum in the user program accordingly. Otherwise the checksum verification will always fail.*
- #define **FLASH_VECTOR_TABLE_SIZE** (0x80)
Total size of the vector table, excluding the bootloader specific checksum.
- #define **FLASH_START_ADDRESS** (**flashLayout[0].sector_start**)
Start address of the bootloader programmable flash.
- #define **FLASH_END_ADDRESS**
End address of the bootloader programmable flash.
- #define **FLASH_PAGE_SIZE** (0x4000) /* flash page size in bytes */
Size of a flash page on the HCS12.
- #define **FLASH_PAGE_OFFSET** (0x8000) /* physical start addr. of pages */
Physical start address of the HCS12 page window.
- #define **FLASH_PPAGE_REG** (*(volatile **blt_int8u** *)0x0015))
PPAGE register to select a specific flash page.
- #define **FLASH_REGS_BASE_ADDRESS** (0x0100)
Base address of the flash related control registers.
- #define **FLASH** ((volatile **tFlashRegs** *)**FLASH_REGS_BASE_ADDRESS**)
Macro for accessing the flash related control registers.
- #define **FLASH_FDIV_MASK** (0x3f)
Bitmask for flash clock divider bits.
- #define **FLASH_FDIV_INVALID** (0xff)
Invalid value for the flash clock divider bits.
- #define **FLASH_CMD_MAX_PARAMS** (4)
Maximum number of flash command parameters.
- #define **FLASH_PHRASE_SIZE** (8)
A phrase is an aligned group of 4 16-bit words, so 8 bytes.
- #define **FLASH_ERASE_SECTOR_CMD** (0x0A)
Erase sector flash command.
- #define **FLASH_PROGRAM_PHRASE_CMD** (0x06)
Program phrase flash command.
- #define **CCIF_BIT** (0x80)
FSTAT - command complete flag bit.
- #define **FDIVLD_BIT** (0x80)
FCLKDIV - clock divider loaded bit.
- #define **ACCERR_BIT** (0x20)
FSTAT - flash access error flag bit.
- #define **FPVIOL_BIT** (0x10)
FSTAT - flash protection violation flag bit.

Typedefs

- `typedef void(* pFlashExeCmdFct) (void)`
Pointer type to flash command execution function.

Functions

- `static blt_bool FlashInitBlock (tFlashBlockInfo *block, blt_addr address)`
Copies data currently in flash to the block->data and sets the base address.
- `static tFlashBlockInfo * FlashSwitchBlock (tFlashBlockInfo *block, blt_addr base_addr)`
Switches blocks by programming the current one and initializing the next.
- `static blt_bool FlashAddToBlock (tFlashBlockInfo *block, blt_addr address, blt_int8u *data, blt_int32u len)`
Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.
- `static blt_bool FlashWriteBlock (tFlashBlockInfo *block)`
Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.
- `static blt_int8u FlashGetGlobalAddrByte (blt_addr addr)`
Reads the byte value from the linear address.
- `static blt_int8u FlashGetPhysPage (blt_addr addr)`
Extracts the physical flash page number from a linear address.
- `static blt_int16u FlashGetPhysAddr (blt_addr addr)`
Extracts the physical address on the flash page number from a linear address.
- `static void FlashExecuteCommand (void)`
Executes the command. The actual code for the command execution is stored as location independant machine code in array flashExecCmd[]. The contents of this array are temporarily copied to RAM. This way the function can be executed from RAM avoiding problem when try to perform a flash operation on the same flash block that this driver is located on.
- `static blt_bool FlashOperate (blt_int8u cmd, blt_addr addr, blt_int16u params[], blt_int8u param_count)`
Prepares the flash command and executes it.
- `void FlashInit (void)`
Initializes the flash driver.
- `blt_bool FlashWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.
- `blt_bool FlashErase (blt_addr addr, blt_int32u len)`
Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.
- `blt_bool FlashWriteChecksum (void)`
Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.
- `blt_bool FlashVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_bool FlashDone (void)`
Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.
- `blt_addr Flash GetUserProgBaseAddress (void)`
Obtains the base address of the flash memory available to the user program. This is basically the last address in the flashLayout table converted to the physical address on the last page, because this is where the address will be in.

Variables

- static const `tFlashSector` `flashLayout` []

If desired, it is possible to set `BOOT_FLASH_CUSTOM_LAYOUT_ENABLE` to > 0 in `blt.conf.h` and then implement your own version of the `flashLayout[]` table in a source-file with the name `flash_layout.c`. This way you customize the flash memory size reserved for the bootloader, without having to modify the `flashLayout[]` table in this file directly. This file will then include `flash_layout.c` so there is no need to compile it additionally with your project.
- static const `tFlashPrescalerSysclockMapping` `flashFDIVlookup` []

Lookup table for determining the flash clock divider setting based on the system clock speed. The flash clock must be around 1MHz and is scaled down from the system clock using a prescaler value. Note that clock speeds in the table are in kHz.
- static const `blt_int8u` `flashExecCmd` []

Array with executable code for performing flash operations.
- static `tFlashBlockInfo` `blockInfo`

Local variable with information about the flash block that is currently being operated on.
- static `tFlashBlockInfo` `bootBlockInfo`

Local variable with information about the flash boot block.
- static `blt_int8u` `flashExecCmdRam` [(`sizeof(flashExecCmd)/sizeof(flashExecCmd[0])`)]

RAM buffer where the executable flash operation code is copied to.

7.333.1 Detailed Description

Bootloader flash driver source file for HCS12 derivatives with error correction code in flash memory, such as the HCS12Pxx. This flash memory uses a different addressing scheme than other HCS12 derivatives.

7.333.2 Function Documentation

7.333.2.1 FlashAddToBlock()

```
static blt_bool FlashAddToBlock (
    tFlashBlockInfo * block,
    blt_addr address,
    blt_int8u * data,
    blt_int32u len ) [static]
```

Programming is done per block. This function adds data to the block that is currently collecting data to be written to flash. If the address is outside of the current block, the current block is written to flash and a new block is initialized.

Parameters

| | |
|----------------------|--|
| <code>block</code> | Pointer to flash block info structure to operate on. |
| <code>address</code> | Flash destination address. |
| <code>data</code> | Pointer to the byte array with data. |
| <code>len</code> | Number of bytes to add to the block. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.333.2.2 FlashDone()

```
blt_bool FlashDone (
    void )
```

Finalizes the flash driver operations. There could still be data in the currently active block that needs to be flashed.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.333.2.3 FlashErase()

```
blt_bool FlashErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the flash memory. Note that this function also checks that no data is erased outside the flash memory region, so the bootloader can never be erased.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.333.2.4 FlashExecuteCommand()

```
static void FlashExecuteCommand (
    void ) [static]
```

Executes the command. The actual code for the command execution is stored as location independant machine code in array flashExecCmd[]. The contents of this array are temporarily copied to RAM. This way the function can be executed from RAM avoiding problem when try to perform a flash operation on the same flash block that this driver is located on.

Returns

none.

Referenced by [FlashOperate\(\)](#).

7.333.2.5 FlashGetGlobalAddrByte()

```
static blt_int8u FlashGetGlobalAddrByte (
    blt_addr addr ) [static]
```

Reads the byte value from the linear address.

Parameters

| | |
|-------------|-----------------|
| <i>addr</i> | Linear address. |
|-------------|-----------------|

Returns

The byte value located at the linear address.

Referenced by [FlashWriteBlock\(\)](#).

7.333.2.6 FlashGetPhysAddr()

```
static blt_int16u FlashGetPhysAddr (
    blt_addr addr ) [static]
```

Extracts the physical address on the flash page number from a linear address.

Parameters

| | |
|-------------|-----------------|
| <i>addr</i> | Linear address. |
|-------------|-----------------|

Returns

The physical address.

Referenced by [FlashGetGlobalAddrByte\(\)](#), [Flash GetUserProgBaseAddress\(\)](#), and [FlashInitBlock\(\)](#).

7.333.2.7 FlashGetPhysPage()

```
static blt_int8u FlashGetPhysPage (
    blt_addr addr ) [static]
```

Extracts the physical flash page number from a linear address.

Parameters

| | |
|-------------------|-----------------|
| <code>addr</code> | Linear address. |
|-------------------|-----------------|

Returns

The page number.

Referenced by [FlashGetGlobalAddrByte\(\)](#), and [FlashInitBlock\(\)](#).

7.333.2.8 Flash GetUserProgBaseAddress()

```
blt_addr Flash GetUserProgBaseAddress (
    void )
```

Obtains the base address of the flash memory available to the user program. This is basically the last address in the flashLayout table converted to the physical address on the last page, because this is where the address will be in.

Obtains the base address of the flash memory available to the user program. This is basically the first address in the flashLayout table.

Returns

Base address.

7.333.2.9 FlashInit()

```
void FlashInit (
    void )
```

Initializes the flash driver.

Returns

none.

7.333.2.10 FlashInitBlock()

```
static blt_bool FlashInitBlock (
    tFlashBlockInfo * block,
    blt_addr address ) [static]
```

Copies data currently in flash to the block->data and sets the base address.

Parameters

| | |
|----------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>address</i> | Base address of the block data. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashAddToBlock\(\)](#).

7.333.2.11 FlashOperate()

```
static blt_bool FlashOperate (
    blt_int8u cmd,
    blt_addr addr,
    blt_int16u params[],
    blt_int8u param_count ) [static]
```

Prepares the flash command and executes it.

Parameters

| | |
|--------------------|---|
| <i>cmd</i> | Command to be launched. |
| <i>addr</i> | Global address to operate on (18-bit). |
| <i>params</i> | Array with additional command parameters. |
| <i>param_count</i> | Number of parameters in the array. |

Returns

BLT_TRUE if operation was successful, otherwise BLT_FALSE.

Referenced by [FlashWriteBlock\(\)](#).

7.333.2.12 FlashSwitchBlock()

```
static tFlashBlockInfo * FlashSwitchBlock (
    tFlashBlockInfo * block,
    blt_addr base_addr ) [static]
```

Switches blocks by programming the current one and initializing the next.

Parameters

| | |
|------------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
| <i>base_addr</i> | Base address of the next block. |

Returns

The pointer of the block info struct that is no being used, or a NULL pointer in case of error.

Referenced by [FlashAddToBlock\(\)](#).

7.333.2.13 FlashVerifyChecksum()

```
blt_bool FlashVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.333.2.14 FlashWrite()

```
blt_bool FlashWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Writes the data to flash through a flash block manager. Note that this function also checks that no data is programmed outside the flash memory region, so the bootloader can never be overwritten.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.333.2.15 FlashWriteBlock()

```
static blt_bool FlashWriteBlock (
    tFlashBlockInfo * block ) [static]
```

Programs FLASH_WRITE_BLOCK_SIZE bytes to flash from the block->data array.

Parameters

| | |
|--------------|--|
| <i>block</i> | Pointer to flash block info structure to operate on. |
|--------------|--|

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [FlashDone\(\)](#).

7.333.2.16 FlashWriteChecksum()

```
blt_bool FlashWriteChecksum (
    void )
```

Writes a checksum of the user program to non-volatile memory. This is performed once the entire user program has been programmed. Through the checksum, the bootloader can check if the programming session was completed, which indicates that a valid user programming is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

7.333.3 Variable Documentation

7.333.3.1 blockInfo

```
tFlashBlockInfo blockInfo [static]
```

Local variable with information about the flash block that is currently being operated on.

The smallest amount of flash that can be programmed is FLASH_WRITE_BLOCK_SIZE. A flash block manager is implemented in this driver and stores info in this variable. Whenever new data should be flashed, it is first added to a RAM buffer, which is part of this variable. Whenever the RAM buffer, which has the size of a flash block, is full or data needs to be written to a different block, the contents of the RAM buffer are programmed to flash. The flash block manager requires some software overhead, yet results in faster flash programming because data is first harvested, ideally until there is enough to program an entire flash block, before the flash device is actually operated on.

Referenced by [FlashDone\(\)](#), and [FlashInit\(\)](#).

7.333.3.2 bootBlockInfo

```
tFlashBlockInfo bootBlockInfo [static]
```

Local variable with information about the flash boot block.

The first block of the user program holds the vector table, which on the STM32 is also the where the checksum is written to. Is it likely that the vector table is first flashed and then, at the end of the programming sequence, the checksum. This means that this flash block need to be written to twice. Normally this is not a problem with flash memory, as long as you write the same values to those bytes that are not supposed to be changed and the locations where you do write to are still in the erased 0xFF state. Unfortunately, writing twice to flash this way, does not work reliably on all micros. This is why we need to have an extra block, the bootblock, placed under the management of the block manager. This way is it possible to implement functionality so that the bootblock is only written to once at the end of the programming sequence.

Referenced by [FlashDone\(\)](#), [FlashInit\(\)](#), and [FlashWriteChecksum\(\)](#).

7.333.3.3 flashExecCmd

```
const blt_int8u flashExecCmd[] [static]
```

Initial value:

```
=  
{  
    0x36,  
    0x34,  
    0xce, 0x01, 0x00,  
    0x1a, 0x06,  
    0x86, 0x80,  
    0x6a, 0x00,  
    0xa7, 0xa7, 0xa7, 0xa7,  
    0x0f, 0x00, 0x80, 0xfc,  
    0x30,  
    0x32,  
    0x3d  
}
```

Array with executable code for performing flash operations.

This array contains the machine code to perform the actual command on the flash device, such as program or erase. the code is compiler and location independent. This allows us to copy it to a ram buffer and execute the code from ram. This way the flash driver can be located in flash memory without running into problems when erasing/programming the same flash block that contains the flash driver. the source code for the machine code is as follows: // launch the command FLASH->fstat = CCIF_BIT; // wait at least 4 cycles (per AN2720) asm("nop"); asm("nop"); asm("nop"); asm("nop"); // wait for command to complete while ((FLASH->fstat & CCIF_BIT) != CCIF_BIT);

Referenced by [FlashDone\(\)](#), and [FlashExecuteCommand\(\)](#).

7.333.3.4 flashLayout

```
const tFlashSector flashLayout[ ] [static]
```

If desired, it is possible to set BOOT_FLASH_CUSTOM_LAYOUT_ENABLE to > 0 in blt_conf.h and then implement your own version of the flashLayout[] table in a source-file with the name flash_layout.c. This way you customize the flash memory size reserved for the bootloader, without having to modify the flashLayout[] table in this file directly. This file will then include flash_layout.c so there is no need to compile it additionally with your project.

Array wit the layout of the flash memory.

Also controls what part of the flash memory is reserved for the bootloader. If the bootloader size changes, the reserved sectors for the bootloader might need adjustment to make sure the bootloader doesn't get overwritten. This layout uses global addresses only. Note that the last part is where the bootloader also resides and it has been entered as 8 chunks of 2kb. This allows flexibility for reserving more/less space for the bootloader in case its size changes in the future.

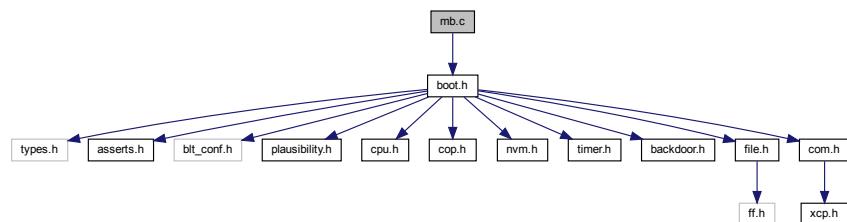
Referenced by [FlashSwitchBlock\(\)](#), [FlashVerifyChecksum\(\)](#), [FlashWrite\(\)](#), and [FlashWriteChecksum\(\)](#).

7.334 mb.c File Reference

Bootloader Modbus communication interface source file.

```
#include "boot.h"
```

Include dependency graph for mb.c:



7.334.1 Detailed Description

Bootloader Modbus communication interface source file.

7.335 mb.h File Reference

Bootloader Modbus communication interface header file.

7.335.1 Detailed Description

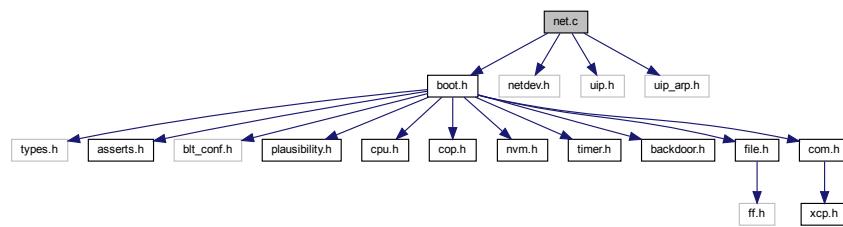
Bootloader Modbus communication interface header file.

7.336 net.c File Reference

Bootloader TCP/IP network communication interface source file.

```
#include "boot.h"
#include "netdev.h"
#include "uip.h"
#include "uip_arp.h"

Include dependency graph for net.c:
```



Macros

- `#define NET_UIP_PERIODIC_TIMER_MS (500)`
Delta time for the uIP periodic timer.
- `#define NET_UIP_ARP_TIMER_MS (10000)`
Delta time for the uIP ARP timer.
- `#define NET_UIP_HEADER_BUF ((struct uip_eth_hdr *)&uip_buf[0])`
Macro for accessing the Ethernet header information in the buffer.

Functions

- `static void NetServerTask (void)`
Runs the TCP/IP server task.
- `void NetInit (void)`
Initializes the TCP/IP network communication interface.
- `void NetTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool NetReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.
- `void NetApp (void)`
The uIP network application that implements XCP on TCP/IP. Note that this application make use of the fact that XCP is request/response based. So no new request will come in when a response is pending for transmission, if so, the transmission of the pending response is aborted.

Variables

- static `blt_int32u` **periodicTimerTimeOut**
Holds the time out value of the uIP periodic timer.
- static `blt_int32u` **ARPTimerTimeOut**
Holds the time out value of the uIP ARP timer.
- static `blt_bool` **netInitializedFlag** = `BLT_FALSE`
Boolean flag to determine if the module was initialized or not.
- static `blt_bool` **netInitializationDeferred** = `BLT_FALSE`
Boolean flag initialized such that the normal initialization via [NetInit\(\)](#) proceeds as usual.

7.336.1 Detailed Description

Bootloader TCP/IP network communication interface source file.

7.336.2 Function Documentation

7.336.2.1 NetApp()

```
void NetApp (
    void )
```

The uIP network application that implements XCP on TCP/IP. Note that this application make use of the fact that XCP is request/response based. So no new request will come in when a response is pending for transmission, if so, the transmission of the pending response is aborted.

Returns

none.

7.336.2.2 NetInit()

```
void NetInit (
    void )
```

Initializes the TCP/IP network communication interface.

Returns

none.

Referenced by [ComInit\(\)](#).

7.336.2.3 NetReceivePacket()

```
blt_bool NetReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

Referenced by [ComTask\(\)](#).

7.336.2.4 NetServerTask()

```
static void NetServerTask (
    void ) [static]
```

Runs the TCP/IP server task.

Returns

none.

Referenced by [NetReceivePacket\(\)](#).

7.336.2.5 NetTransmitPacket()

```
void NetTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

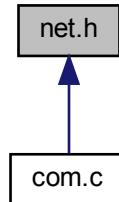
none.

Referenced by [ComTransmitPacket\(\)](#).

7.337 net.h File Reference

Bootloader TCP/IP network communication interface header file.

This graph shows which files directly or indirectly include this file:



Data Structures

- struct `uip_tcp_appstate_t`

Define the `uip_tcp_appstate_t` datatype. This is the state of our tcp/ip application, and the memory required for this state is allocated together with each TCP connection. One application state for each TCP connection.

Functions

- void `NetInit` (void)
Initializes the TCP/IP network communication interface.
- void `NetApp` (void)
The uIP network application that implements XCP on TCP/IP. Note that this application make use of the fact that XCP is request/response based. So no new request will come in when a response is pending for transmission, if so, the transmission of the pending response is aborted.
- void `NetTransmitPacket` (`blt_int8u` *data, `blt_int8u` len)
Transmits a packet formatted for the communication interface.
- `blt_bool` `NetReceivePacket` (`blt_int8u` *data, `blt_int8u` *len)
Receives a communication interface packet if one is present.

7.337.1 Detailed Description

Bootloader TCP/IP network communication interface header file.

7.337.2 Function Documentation

7.337.2.1 NetApp()

```
void NetApp (
    void )
```

The uIP network application that implements XCP on TCP/IP. Note that this application make use of the fact that XCP is request/response based. So no new request will come in when a response is pending for transmission, if so, the transmission of the pending response is aborted.

Returns

none.

7.337.2.2 NetInit()

```
void NetInit (
    void )
```

Initializes the TCP/IP network communication interface.

Returns

none.

Referenced by [ComInit\(\)](#).

7.337.2.3 NetReceivePacket()

```
blt_bool NetReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

Referenced by [ComTask\(\)](#).

7.337.2.4 NetTransmitPacket()

```
void NetTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

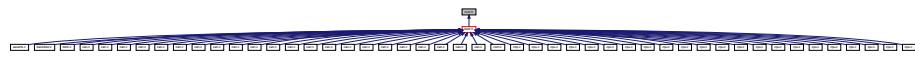
none.

Referenced by [ComTransmitPacket\(\)](#).

7.338 nvm.h File Reference

Bootloader non-volatile memory driver header file.

This graph shows which files directly or indirectly include this file:



Macros

- `#define BLT_NVM_ERROR (0x00)`
Return code for success.
- `#define BLT_NVM_OKAY (0x01)`
Return code for error.
- `#define BLT_NVM_NOT_IN_RANGE (0x02)`
Return code for not in range.

Functions

- `void NvmInit (void)`
Initializes the NVM driver.
- `blt_bool NvmWrite (blt_addr addr, blt_int32u len, blt_int8u *data)`
Programs the non-volatile memory.
- `blt_bool NvmErase (blt_addr addr, blt_int32u len)`
Erases the non-volatile memory.
- `blt_bool NvmVerifyChecksum (void)`
Verifies the checksum, which indicates that a valid user program is present and can be started.
- `blt_addr Nvm GetUserProgBaseAddress (void)`
Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.
- `blt_bool NvmDone (void)`
Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

7.338.1 Detailed Description

Bootloader non-volatile memory driver header file.

7.338.2 Function Documentation

7.338.2.1 NvmDone()

```
blt_bool NvmDone (
    void )
```

Once all erase and programming operations are completed, this function is called, so at the end of the programming session and right before a software reset is performed. It is used to calculate a checksum and program this into flash. This checksum is later used to determine if a valid user program is present in flash.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [XcpCmdProgram\(\)](#).

7.338.2.2 NvmErase()

```
blt_bool NvmErase (
    blt_addr addr,
    blt_int32u len )
```

Erases the non-volatile memory.

Parameters

| | |
|-------------|------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [XcpCmdProgramClear\(\)](#).

7.338.2.3 Nvm GetUserProgBaseAddress()

```
blt_addr Nvm GetUserProgBaseAddress (
    void )
```

Obtains the base address of the non-volatile memory available to the user program. This is typically that start of the vector table.

Returns

Base address.

7.338.2.4 NvmInit()

```
void NvmInit (
    void )
```

Initializes the NVM driver.

Returns

none.

Referenced by [BootInit\(\)](#), [FileTask\(\)](#), and [XcpCmdConnect\(\)](#).

7.338.2.5 NvmVerifyChecksum()

```
blt_bool NvmVerifyChecksum (
    void )
```

Verifies the checksum, which indicates that a valid user program is present and can be started.

Returns

BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [CpuStartUserProgram\(\)](#).

7.338.2.6 NvmWrite()

```
blt_bool NvmWrite (
    blt_addr addr,
    blt_int32u len,
    blt_int8u * data )
```

Programs the non-volatile memory.

Parameters

| | |
|-------------|-----------------------------|
| <i>addr</i> | Start address. |
| <i>len</i> | Length in bytes. |
| <i>data</i> | Pointer to the data buffer. |

Returns

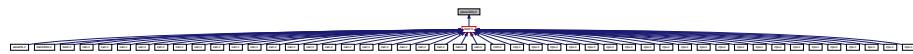
BLT_TRUE if successful, BLT_FALSE otherwise.

Referenced by [XcpCmdProgram\(\)](#), and [XcpCmdProgramMax\(\)](#).

7.339 plausibility.h File Reference

Bootloader plausibility check header file, for checking the configuration at compile time.

This graph shows which files directly or indirectly include this file:



7.339.1 Detailed Description

Bootloader plausibility check header file, for checking the configuration at compile time.

7.340 rs232.h File Reference

Bootloader RS232 communication interface header file.

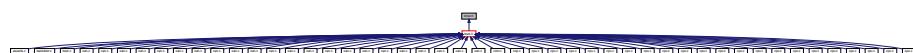
7.340.1 Detailed Description

Bootloader RS232 communication interface header file.

7.341 timer.h File Reference

Bootloader timer driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- void [TimerInit](#) (void)
Initializes the polling based millisecond timer driver.
- void [TimerUpdate](#) (void)
Updates the millisecond timer.
- [blt_int32u TimerGet](#) (void)
Obtains the counter value of the millisecond timer.
- void [TimerReset](#) (void)
Reset the timer by placing the timer back into it's default reset configuration.

7.341.1 Detailed Description

Bootloader timer driver header file.

7.341.2 Function Documentation

7.341.2.1 TimerGet()

```
blt_int32u TimerGet (
    void )
```

Obtains the counter value of the millisecond timer.

Returns

Current value of the millisecond timer.

Referenced by [BackDoorCheck\(\)](#), [CanDisabledModeEnter\(\)](#), [CanDisabledModeExit\(\)](#), [CanFreezeModeEnter\(\)](#), [CanFreezeModeExit\(\)](#), [CanInit\(\)](#), [CanTransmitPacket\(\)](#), [FlashEraseSectors\(\)](#), [FlashWriteBlock\(\)](#), [HAL_GetTick\(\)](#), [NetInit\(\)](#), and [NetServerTask\(\)](#).

7.341.2.2 TimerInit()

```
void TimerInit (
    void )
```

Initializes the polling based millisecond timer driver.

Returns

none.

Ideally a 100 kHz free running counter is used as the foundation for the timer modules, as this gives 10us ticks that can be reused by other modules. The S32K11 timers unfortunately do not offer a flexible prescaler for their timers to realize such a 100 kHz free running counter. For this reason, the SysTick counter is used instead. Its 24-bit free running down-counter runs at the system speed.

Returns

none.

Attention

To keep the ROM footprint low, this function aims to only use LL driver inline functions.

Returns

none.

Ideally a 100 kHz free running counter is used as the foundation for the timer modules, as this gives 10us ticks that can be reused by other modules. The XMC1 timers unfortunately do not offer a flexible prescaler for their timers to realize such a 100 kHz free running counter. For this reason, the SysTick counter is used instead. Its 24-bit free running down-counter runs at the system speed.

Returns

none.

Ideally a 100 kHz free running counter is used as the foundation for the timer modules, as this gives 10us ticks that can be reused by other modules. The S32K14 timers unfortunately do not offer a flexible prescaler for their timers to realize such a 100 kHz free running counter. For this reason, the SysTick counter is used instead. Its 24-bit free running down-counter runs at the system speed.

Returns

none.

Ideally a 100 kHz free running counter is used as the foundation for the timer modules, as this gives 10us ticks that can be reused by other modules. The XMC4 timers unfortunately do not offer a flexible prescaler for their timers to realize such a 100 kHz free running counter. For this reason, the SysTick counter is used instead. Its 24-bit free running down-counter runs at the system speed.

Returns

none.

Referenced by [BootInit\(\)](#).

7.341.2.3 TimerReset()

```
void TimerReset (
    void )
```

Reset the timer by placing the timer back into it's default reset configuration.

Returns

none.

Referenced by [TimerInit\(\)](#).

7.341.2.4 TimerUpdate()

```
void TimerUpdate (
    void )
```

Updates the millisecond timer.

Returns

none.

Referenced by [BootTask\(\)](#), and [TimerGet\(\)](#).

7.342 ram_func.h File Reference

RAM function macros header file.

Macros

- `#define BLT_RAM_FUNC_BEGIN _Pragma("section code cpu0_psram")`
Macro used at the start of a function implementation to inform the linker that the function should be copied from flash to RAM by the startup code and also run from RAM.
- `#define BLT_RAM_FUNC_END _Pragma("section code restore")`
Macro used at the end of a function implementation to inform the linker that the RAM function is now ended.

7.342.1 Detailed Description

RAM function macros header file.

7.343 ram_func.h File Reference

RAM function macros header file.

Macros

- `#define BLT_RAM_FUNC_BEGIN _Pragma("section code cpu0_psram")`
Macro used at the start of a function implementation to inform the linker that the function should be copied from flash to RAM by the startup code and also run from RAM.
- `#define BLT_RAM_FUNC_END _Pragma("section code restore")`
Macro used at the end of a function implementation to inform the linker that the RAM function is now ended.

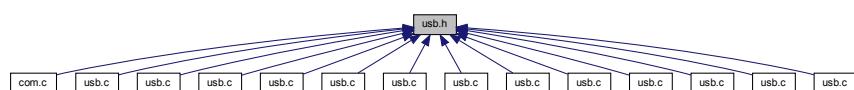
7.343.1 Detailed Description

RAM function macros header file.

7.344 usb.h File Reference

Bootloader USB communication interface header file.

This graph shows which files directly or indirectly include this file:



Functions

- `void UsbInit (void)`
Initializes the USB communication interface.
- `void UsbFree (void)`
Releases the USB communication interface.
- `void UsbTransmitPacket (blt_int8u *data, blt_int8u len)`
Transmits a packet formatted for the communication interface.
- `blt_bool UsbReceivePacket (blt_int8u *data, blt_int8u *len)`
Receives a communication interface packet if one is present.

7.344.1 Detailed Description

Bootloader USB communication interface header file.

7.344.2 Function Documentation

7.344.2.1 UsbFree()

```
void UsbFree (
    void )
```

Releases the USB communication interface.

Returns

none.

Referenced by [ComFree\(\)](#).

7.344.2.2 UsbInit()

```
void UsbInit (
    void )
```

Initializes the USB communication interface.

Returns

none.

Referenced by [ComInit\(\)](#).

7.344.2.3 UsbReceivePacket()

```
blt_bool UsbReceivePacket (
    blt_int8u * data,
    blt_int8u * len )
```

Receives a communication interface packet if one is present.

Parameters

| | |
|-------------|---|
| <i>data</i> | Pointer to byte array where the data is to be stored. |
| <i>len</i> | Pointer where the length of the packet is to be stored. |

Returns

BLT_TRUE if a packet was received, BLT_FALSE otherwise.

Referenced by [ComTask\(\)](#).

7.344.2.4 UsbTransmitPacket()

```
void UsbTransmitPacket (
    blt_int8u * data,
    blt_int8u len )
```

Transmits a packet formatted for the communication interface.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte array with data that it to be transmitted. |
| <i>len</i> | Number of bytes that are to be transmitted. |

Returns

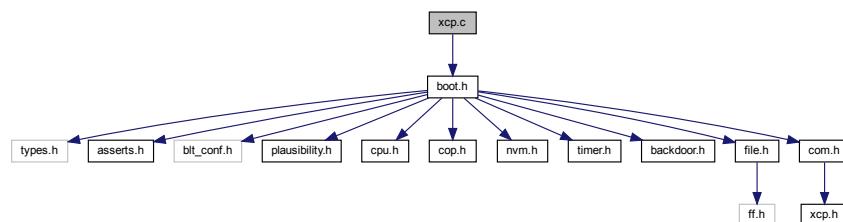
none.

Referenced by [ComTransmitPacket\(\)](#).

7.345 xcp.c File Reference

XCP 1.0 protocol core source file.

```
#include "boot.h"
Include dependency graph for xcp.c:
```



Data Structures

- struct `tXcpInfo`
Structure type for grouping XCP internal module information.

Functions

- static void `XcpTransmitPacket (blt_int8u *data, blt_int16s len)`
Transmits the packet using the xcp transport layer.
- static `blt_int8u XcpComputeChecksum (blt_int32u address, blt_int32u length, blt_int32u *checksum)`
Called by the BUILD_CHECKSUM command to perform a checksum calculation over the specified memory region.
- static `blt_int8u XcpGetSeed (blt_int8u resource, blt_int8u *seed)`
Provides a seed to the XCP master that will be used for the key generation when the master attempts to unlock the specified resource. Called by the GET_SEED command.
- static `blt_int8u XcpVerifyKey (blt_int8u resource, blt_int8u *key, blt_int8u len)`
Called by the UNLOCK command and checks if the key to unlock the specified resource was correct. If so, then the resource protection will be removed.
- static void `XcpProtectResources (void)`
Utility function to protects all the available resources.
- static void `XcpSetCtoError (blt_int8u error)`
Prepares the cto packet data for the specified error.
- static `blt_int32u XcpGetOrderedLong (blt_int8u const *data)`
Obtains a 32-bit value from a byte buffer taking into account Intel or Motorola byte ordering.
- static void `XcpSetOrderedLong (blt_int32u value, blt_int8u *data)`
Stores a 32-bit value into a byte buffer taking into account Intel or Motorola byte ordering.
- static void `XcpCmdConnect (blt_int8u *data)`
XCP command processor function which handles the CONNECT command as defined by the protocol.
- static void `XcpCmdDisconnect (blt_int8u *data)`
XCP command processor function which handles the DISCONNECT command as defined by the protocol.
- static void `XcpCmdGetStatus (blt_int8u *data)`
XCP command processor function which handles the GET_STATUS command as defined by the protocol.
- static void `XcpCmdSynch (blt_int8u *data)`
XCP command processor function which handles the SYNCH command as defined by the protocol.
- static void `XcpCmdGetId (blt_int8u *data)`
XCP command processor function which handles the GET_ID command as defined by the protocol.
- static void `XcpCmdSetMta (blt_int8u *data)`
XCP command processor function which handles the SET_MTA command as defined by the protocol.
- static void `XcpCmdUpload (blt_int8u *data)`
XCP command processor function which handles the UPLOAD command as defined by the protocol.
- static void `XcpCmdShortUpload (blt_int8u *data)`
XCP command processor function which handles the SHORT_UPLOAD command as defined by the protocol.
- static void `XcpCmdBuildCheckSum (blt_int8u *data)`
XCP command processor function which handles the BUILD_CHECKSUM command as defined by the protocol.
- static void `XcpCmdGetSeed (blt_int8u *data)`
XCP command processor function which handles the GET_SEED command as defined by the protocol.
- static void `XcpCmdUnlock (blt_int8u *data)`
XCP command processor function which handles the UNLOCK command as defined by the protocol.
- static void `XcpCmdProgramMax (blt_int8u *data)`
XCP command processor function which handles the PROGRAM_MAX command as defined by the protocol.
- static void `XcpCmdProgram (blt_int8u *data)`
XCP command processor function which handles the PROGRAM command as defined by the protocol.

- static void [XcpCmdProgramStart \(blt_int8u *data\)](#)
XCP command processor function which handles the PROGRAM_START command as defined by the protocol.
- static void [XcpCmdProgramClear \(blt_int8u *data\)](#)
XCP command processor function which handles the PROGRAM_CLEAR command as defined by the protocol.
- static void [XcpCmdProgramReset \(blt_int8u *data\)](#)
XCP command processor function which handles the PROGRAM_RESET command as defined by the protocol.
- static void [XcpCmdProgramPrepare \(blt_int8u *data\)](#)
XCP command processor function which handles the PROGRAM_PREPARE command as defined by the protocol.
- void [XcpInit \(void\)](#)
Initializes the XCP driver. Should be called once upon system startup.
- [blt_bool XcpIsConnected \(void\)](#)
Obtains information about the XCP connection state.
- void [XcpPacketTransmitted \(void\)](#)
Informs the core that a pending packet transmission was completed by the transport layer.
- void [XcpPacketReceived \(blt_int8u *data, blt_int8u len\)](#)
Informs the core that a new packet was received by the transport layer.

Variables

- static const [blt_int8s xcpStationId \[\] = XCP_STATION_ID_STRING](#)
String buffer with station id.
- static [tXcpInfo xcpInfo](#)
Local variable for storing XCP internal module info.

7.345.1 Detailed Description

XCP 1.0 protocol core source file.

7.345.2 Function Documentation

7.345.2.1 XcpCmdBuildCheckSum()

```
static void XcpCmdBuildCheckSum (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the BUILD_CHECKSUM command as defined by the protocol.

Parameters

| | |
|-------------------|--|
| <code>data</code> | Pointer to a byte buffer with the packet data. |
|-------------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.2 XcpCmdConnect()

```
static void XcpCmdConnect (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the CONNECT command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.3 XcpCmdDisconnect()

```
static void XcpCmdDisconnect (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the DISCONNECT command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.4 XcpCmdGetId()

```
static void XcpCmdGetId (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the GET_ID command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).**7.345.2.5 XcpCmdGetSeed()**

```
static void XcpCmdGetSeed (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the GET_SEED command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).**7.345.2.6 XcpCmdGetStatus()**

```
static void XcpCmdGetStatus (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the GET_STATUS command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.7 XcpCmdProgram()

```
static void XcpCmdProgram (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the PROGRAM command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.8 XcpCmdProgramClear()

```
static void XcpCmdProgramClear (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the PROGRAM_CLEAR command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.9 XcpCmdProgramMax()

```
static void XcpCmdProgramMax (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the PROGRAM_MAX command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.10 XcpCmdProgramPrepare()

```
static void XcpCmdProgramPrepare (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the PROGRAM_PREPARE command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.11 XcpCmdProgramReset()

```
static void XcpCmdProgramReset (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the PROGRAM_RESET command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.12 XcpCmdProgramStart()

```
static void XcpCmdProgramStart (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the PROGRAM_START command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.13 XcpCmdSetMta()

```
static void XcpCmdSetMta (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the SET_MTA command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.14 XcpCmdShortUpload()

```
static void XcpCmdShortUpload (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the SHORT_UPLOAD command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.15 XcpCmdSynch()

```
static void XcpCmdSynch (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the SYNCH command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.16 XcpCmdUnlock()

```
static void XcpCmdUnlock (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the UNLOCK command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.17 XcpCmdUpload()

```
static void XcpCmdUpload (
    blt_int8u * data ) [static]
```

XCP command processor function which handles the UPLOAD command as defined by the protocol.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to a byte buffer with the packet data. |
|-------------|--|

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.18 XcpComputeChecksum()

```
static blt_int8u XcpComputeChecksum (
    blt_int32u address,
    blt_int32u length,
    blt_int32u * checksum ) [static]
```

Called by the BUILD_CHECKSUM command to perform a checksum calculation over the specified memory region.

Parameters

| | |
|-----------------|---|
| <i>address</i> | The start address of the memory region. |
| <i>length</i> | Length of the memory region in bytes. |
| <i>checksum</i> | Pointer to where the calculated checksum is to be stored. |

Returns

Checksum type that was used during the checksum calculation.

Referenced by [XcpCmdBuildCheckSum\(\)](#).

7.345.2.19 XcpGetOrderedLong()

```
static blt_int32u XcpGetOrderedLong (
    blt_int8u const * data ) [static]
```

Obtains a 32-bit value from a byte buffer taking into account Intel or Motorola byte ordering.

Parameters

| | |
|-------------|--|
| <i>data</i> | Array to the buffer with the 32-bit value stored as bytes. |
|-------------|--|

Returns

The 32-bit value.

Referenced by [XcpCmdBuildCheckSum\(\)](#), [XcpCmdProgramClear\(\)](#), [XcpCmdSetMta\(\)](#), and [XcpCmdShortUpload\(\)](#).

7.345.2.20 XcpGetSeed()

```
static blt_int8u XcpGetSeed (
    blt_int8u resource,
    blt_int8u * seed ) [static]
```

Provides a seed to the XCP master that will be used for the key generation when the master attempts to unlock the specified resource. Called by the GET_SEED command.

Parameters

| | |
|-----------------|--|
| <i>resource</i> | Resource that the seed is requested for (XCP_RES_XXX). |
| <i>seed</i> | Pointer to byte buffer where the seed will be stored. |

Returns

Length of the seed in bytes.

Referenced by [XcpCmdGetSeed\(\)](#).

7.345.2.21 XcpInit()

```
void XcpInit (
    void )
```

Initializes the XCP driver. Should be called once upon system startup.

Returns

none

Referenced by [ComInit\(\)](#).

7.345.2.22 XcpIsConnected()

```
blt_bool XcpIsConnected (
    void )
```

Obtains information about the XCP connection state.

Returns

BLT_TRUE if an XCP connection is established, BLT_FALSE otherwise.

Referenced by [ComIsConnected\(\)](#).

7.345.2.23 XcpPacketReceived()

```
void XcpPacketReceived (
    blt_int8u * data,
    blt_int8u len )
```

Informs the core that a new packet was received by the transport layer.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte buffer with packet data. |
| <i>len</i> | Number of bytes in the packet. |

Returns

none

Referenced by [ComTask\(\)](#), and [NetApp\(\)](#).

7.345.2.24 XcpPacketTransmitted()

```
void XcpPacketTransmitted (
    void )
```

Informs the core that a pending packet transmission was completed by the transport layer.

Returns

none

Referenced by [ComTransmitPacket\(\)](#).

7.345.2.25 XcpProtectResources()

```
static void XcpProtectResources (
    void ) [static]
```

Utility function to protects all the available resources.

Returns

none

Referenced by [XcpCmdConnect\(\)](#), [XcpCmdDisconnect\(\)](#), and [XcpCmdUnlock\(\)](#).

7.345.2.26 XcpSetCtoError()

```
static void XcpSetCtoError (
    blt_int8u error ) [static]
```

Prepares the cto packet data for the specified error.

Parameters

| | |
|--------------------|-------------------------------|
| <code>error</code> | XCP error code (XCP_ERR_XXX). |
|--------------------|-------------------------------|

Returns

none

Referenced by [XcpCmdConnect\(\)](#), [XcpCmdGetSeed\(\)](#), [XcpCmdProgram\(\)](#), [XcpCmdProgramClear\(\)](#), [XcpCmdProgramMax\(\)](#), [XcpCmdProgramPrepare\(\)](#), [XcpCmdProgramReset\(\)](#), [XcpCmdProgramStart\(\)](#), [XcpCmdShortUpload\(\)](#), [XcpCmdSynch\(\)](#), [XcpCmdUnlock\(\)](#), [XcpCmdUpload\(\)](#), and [XcpPacketReceived\(\)](#).

7.345.2.27 XcpSetOrderedLong()

```
static void XcpSetOrderedLong (
    blt_int32u value,
    blt_int8u * data ) [static]
```

Stores a 32-bit value into a byte buffer taking into account Intel or Motorola byte ordering.

Parameters

| | |
|--------------------|--|
| <code>value</code> | The 32-bit value to store in the buffer. |
| <code>data</code> | Array to the buffer for storage. |

Referenced by [XcpCmdBuildCheckSum\(\)](#), and [XcpCmdGetId\(\)](#).

7.345.2.28 XcpTransmitPacket()

```
static void XcpTransmitPacket (
    blt_int8u * data,
    blt_int16s len ) [static]
```

Transmits the packet using the xcp transport layer.

Parameters

| | |
|-------------------|---|
| <code>data</code> | Pointer to the byte buffer with packet data. |
| <code>len</code> | Number of data bytes that need to be transmitted. |

Returns

none

Referenced by [XcpPacketReceived\(\)](#).

7.345.2.29 XcpVerifyKey()

```
static blt_int8u XcpVerifyKey (
    blt_int8u resource,
    blt_int8u * key,
    blt_int8u len ) [static]
```

Called by the UNLOCK command and checks if the key to unlock the specified resource was correct. If so, then the resource protection will be removed.

Parameters

| | |
|-----------------|---|
| <i>resource</i> | resource to unlock (XCP_RES_XXX). |
| <i>key</i> | pointer to the byte buffer holding the key. |
| <i>len</i> | length of the key in bytes. |

Returns

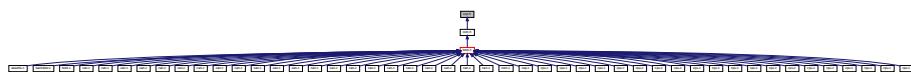
1 if the key was correct, 0 otherwise.

Referenced by [XcpCmdUnlock\(\)](#).

7.346 xcp.h File Reference

XCP 1.0 protocol core header file.

This graph shows which files directly or indirectly include this file:



Macros

- #define **XCP_CTO_PACKET_LEN** ([ComGetActiveInterfaceMaxRxLen\(\)](#))

Maximum length of the transport layer's command transmit object packet.
- #define **XCP.DTO_PACKET_LEN** ([ComGetActiveInterfaceMaxTxLen\(\)](#))

Maximum length of the transport layer's data transmit object packet.
- #define **XCP_STATION_ID_STRING** "OpenBLT"

Name in string format that is used to identify the ECU to the XCP master using the GET_ID command.
- #define **XCP_MOTOROLA_FORMAT** (0x00)

XCP byte ordering according to the Intel (little-endian).
- #define **XCP_RES_CALIBRATION_EN** (0)

Enable (=1) or disable (=0) support for the calibration resource. This is required when data is written to RAM during the XCP session.
- #define **XCP_RES_PAGING_EN** (0)

Enable (=1) or disable (=0) support for the paging resource. This is required when switching between application specific calibration pages should be supported. In this case the application specific external functions AppCalSetPage and AppCalGetPage must be provided.

- **#define XCP_RES_PROGRAMMING_EN (1)**
Enable (=1) or disable (=0) support for the programming resource. This is required when non-volatile memory will be erased or programmed during an XCP session. In this case the following external hardware specific functions must be provided: NvmWrite, NvmErase and CpuStartUserProgram.
- **#define XCP_RES_DATA_ACQUISITION_EN (0)**
Enable (=1) or disable (=0) support for the data acquisition resource. This note that this feature is currently not supported by the XCP driver.
- **#define XCP_RES_DATA_STIMULATION_EN (0)**
Enable (=1) or disable (=0) support for the data stimulation resource. This note that this feature is currently not supported by the XCP driver.
- **#define XCP_SEED_KEY_PROTECTION_EN (1)**
Enable (=1) or disable (=0) support for the seed/key protection feature. If enabled, the XCP master has to perform a GET_SEED/UNLOCK sequence to obtain access to a resource. The protection algorithm is implemented in Xcp↔GetSeed and XcpVerifyKey.
- **#define XCP_UPLOAD_EN (0)**
Enable (=1) or disable (=0) uploading. By default, XCP always allows memory read operations using the commands UPLOAD and SHORT_UPLOAD. This is not always desired for security reasons. If disabled, memory reads via XCP always return zero values.
- **#define XCP_PACKET RECEIVED HOOK_EN (0)**
Enable (=1) or disable the hook function that gets called each time an XCP packet was received from the host.
- **#define XCP_VERSION_PROTOCOL_LAYER (0x0100)**
XCP protocol layer version number (16-bit).
- **#define XCP_VERSION_TRANSPORT_LAYER (0x0100)**
XCP transport layer version number (16-bit).
- **#define XCP_PID_RES (0xff)**
Command response packet identifier.
- **#define XCP_PID_ERR (0xfe)**
Error packet identifier.
- **#define XCP_ERR_CMD_SYNCH (0x00)**
Cmd processor synchronization error code.
- **#define XCP_ERR_CMD_BUSY (0x10)**
Command was not executed error code.
- **#define XCP_ERR_CMD_UNKNOWN (0x20)**
Unknown or unsupported command error code.
- **#define XCP_ERR_OUT_OF_RANGE (0x22)**
Parameter out of range error code.
- **#define XCP_ERR_ACCESS_LOCKED (0x25)**
Protected error code. Seed/key required.
- **#define XCP_ERR_PAGE_NOT_VALID (0x26)**
Cal page not valid error code.
- **#define XCP_ERR_SEQUENCE (0x29)**
Sequence error code.
- **#define XCP_ERR_GENERIC (0x31)**
Generic error code.
- **#define XCP_CMD_CONNECT (0xff)**
CONNECT command code.
- **#define XCP_CMD_DISCONNECT (0xfe)**
DISCONNECT command code.
- **#define XCP_CMD_GET_STATUS (0xfd)**
GET_STATUS command code.
- **#define XCP_CMD_SYNCH (0xfc)**
SYNCH command code.

- #define **XCP_CMD_GET_ID** (0xfa)
GET_ID command code.
- #define **XCP_CMD_GET_SEED** (0xf8)
GET_SEED command code.
- #define **XCP_CMD_UNLOCK** (0xf7)
UNLOCK command code.
- #define **XCP_CMD_SET_MTA** (0xf6)
SET_MTA command code.
- #define **XCP_CMD_UPLOAD** (0xf5)
UPLOAD command code.
- #define **XCP_CMD_SHORT_UPLOAD** (0xf4)
SHORT_UPLOAD command code.
- #define **XCP_CMD_BUILD_CHECKSUM** (0xf3)
BUILD_CHECKSUM command code.
- #define **XCP_CMD_DOWNLOAD** (0xf0)
DOWNLOAD command code.
- #define **XCP_CMD_DOWLOAD_MAX** (0xee)
DOWNLOAD_MAX command code.
- #define **XCP_CMD_SET_CAL_PAGE** (0xeb)
SET_CALPAGE command code.
- #define **XCP_CMD_GET_CAL_PAGE** (0xea)
GET_CALPAGE command code.
- #define **XCP_CMD_PROGRAM_START** (0xd2)
PROGRAM_START command code.
- #define **XCP_CMD_PROGRAM_CLEAR** (0xd1)
PROGRAM_CLEAR command code.
- #define **XCP_CMD_PROGRAM** (0xd0)
PROGRAM command code.
- #define **XCP_CMD_PROGRAM_RESET** (0xcf)
PROGRAM_RESET command code.
- #define **XCP_CMD_PROGRAM_PREPARE** (0xcc)
PROGRAM_PREPARE command code.
- #define **XCP_CMD_PROGRAM_MAX** (0xc9)
PROGRAM_MAX command code.
- #define **XCP_RES_PGM** (0x10)
ProGramming resource.
- #define **XCP_RES_STIM** (0x08)
data STIMulation resource.
- #define **XCP_RES_DAQ** (0x04)
Data AcQuisition resource.
- #define **XCP_RES_CALPAG** (0x01)
CALibration and PAGing resource.
- #define **XCP_CS_ADD11** (0x01)
Add byte into byte checksum.
- #define **XCP_CS_ADD12** (0x02)
Add byte into word checksum.
- #define **XCP_CS_ADD14** (0x03)
Add byte into dword checksum.
- #define **XCP_CS_ADD22** (0x04)
Add word into word checksum.
- #define **XCP_CS_ADD24** (0x05)

- `#define XCP_CS_ADD44 (0x06)`

Add word into dword checksum.
- `#define XCP_CS_CRC16 (0x07)`

Add dword into dword checksum.
- `#define XCP_CS_CRC16CITT (0x08)`

Use 16-bit CRC algorithm.
- `#define XCP_CS_CRC32 (0x09)`

Use 32-bit CRC algorithm.
- `#define XCP_CS_USER (0xff)`

Use user defined algorithm.
- `#define XCP_SEED_MAX_LEN (BOOT_XCP_SEED_MAX_LEN)`

Maximum number of bytes of a seed for the seed/key security feature.
- `#define XCP_KEY_MAX_LEN (BOOT_XCP_KEY_MAX_LEN)`

Maximum number of bytes of a key for the seed/key security feature.

Functions

- `void XcpInit (void)`

Initializes the XCP driver. Should be called once upon system startup.
- `blt_bool XcpIsConnected (void)`

Obtains information about the XCP connection state.
- `void XcpPacketTransmitted (void)`

Informs the core that a pending packet transmission was completed by the transport layer.
- `void XcpPacketReceived (blt_int8u *data, blt_int8u len)`

Informs the core that a new packet was received by the transport layer.

7.346.1 Detailed Description

XCP 1.0 protocol core header file.

7.346.2 Macro Definition Documentation

7.346.2.1 XCP_PACKET RECEIVED HOOK EN

```
#define XCP_PACKET_RECEIVED_HOOK_EN (0)
```

Enable (=1) or disable the hook function that gets called each time an XCP packet was received from the host.

A master-slave bootloader can be realized by using this hook-function. The mode parameter in the XCP Connect command can be interpreted as a node ID. When trying to connect to a slave, a gateway could be activated that passes the packet on to the slave. When the response packet is received from the slave, `ComTransmitPacket()` can be called to pass the response on to the host. At the end of a firmware update procedure, the XCP Program Reset command is called, which can be used to deactivate the gateway. If this hook-function returns BLT_TRUE, the packet is no longer processed by the XCP module. If it returns BLT_FALSE, then the packet is processed as usual by the XCP module.

7.346.3 Function Documentation

7.346.3.1 XcpInit()

```
void XcpInit (
    void )
```

Initializes the XCP driver. Should be called once upon system startup.

Returns

none

Referenced by [ComInit\(\)](#).

7.346.3.2 XcpIsConnected()

```
blt_bool XcpIsConnected (
    void )
```

Obtains information about the XCP connection state.

Returns

BLT_TRUE is an XCP connection is established, BLT_FALSE otherwise.

Referenced by [ComIsConnected\(\)](#).

7.346.3.3 XcpPacketReceived()

```
void XcpPacketReceived (
    blt_int8u * data,
    blt_int8u len )
```

Informs the core that a new packet was received by the transport layer.

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to byte buffer with packet data. |
| <i>len</i> | Number of bytes in the packet. |

Returns

none

Referenced by [ComTask\(\)](#), and [NetApp\(\)](#).

7.346.3.4 XcpPacketTransmitted()

```
void XcpPacketTransmitted (
    void )
```

Informs the core that a pending packet transmission was completed by the transport layer.

Returns

none

Referenced by [ComTransmitPacket\(\)](#).

Index

_template/GCC/cpu_comp.c
 CpuIrqDisable, 573
 CpuIrqEnable, 573
_template/can.c
 CanGetSpeedConfig, 84
 CanInit, 84
 CanReceivePacket, 85
 canTiming, 85
 CanTransmitPacket, 85
_template/cpu.c
 CpuInit, 160
 CpuMemcpy, 160
 CpuMemSet, 161
 CpuStartUserProgram, 161
_template/flash.c
 blockInfo, 230
 bootBlockInfo, 231
 FlashAddToBlock, 226
 FlashDone, 226
 FlashErase, 226
 FlashEraseSectors, 227
 FlashGetSectorIdx, 227
 Flash GetUserProgBaseAddress, 227
 FlashInit, 228
 FlashInitBlock, 228
 flashLayout, 231
 FlashSwitchBlock, 228
 FlashVerifyChecksum, 229
 FlashWrite, 229
 FlashWriteBlock, 230
 FlashWriteChecksum, 230
_template/flash.h
 FlashDone, 465
 FlashErase, 466
 Flash GetUserProgBaseAddress, 466
 FlashInit, 467
 FlashVerifyChecksum, 467
 FlashWrite, 467
 FlashWriteChecksum, 468
_template/nvm.c
 NvmDone, 670
 NvmErase, 670
 Nvm GetUserProgBaseAddress, 671
 NvmInit, 671
 NvmVerifyChecksum, 671
 NvmWrite, 671
_template/timer.c
 TimerGet, 765
 TimerInit, 765
 TimerReset, 766
 TimerUpdate, 766
_template/types.h
 blt_addr, 831
 blt_bool, 831
 blt_char, 831
 blt_int16s, 832
 blt_int16u, 832
 blt_int32s, 832
 blt_int32u, 832
 blt_int8s, 832
 blt_int8u, 832
_template/usb.c
 UsbFifoMgrCreate, 890
 UsbFifoMgrInit, 890
 UsbFifoMgrRead, 891
 UsbFifoMgrScan, 891
 UsbFifoMgrWrite, 891
 UsbFree, 892
 UsbInit, 892
 UsbReceiveByte, 892
 UsbReceivePacket, 893
 UsbReceivePipeBulkOUT, 893
 UsbTransmitByte, 893
 UsbTransmitPacket, 894
 UsbTransmitPipeBulkIN, 894
address
 tSrecLineParseObject, 75
ARMCM0_S32K11/can.c
 CanDisabledModeEnter, 88
 CanDisabledModeExit, 88
 CanFreezeModeEnter, 88
 CanFreezeModeExit, 88
 CanGetSpeedConfig, 89
 CanInit, 89
 CanReceivePacket, 89
 canTiming, 90
 CanTransmitPacket, 90
ARMCM0_S32K11/cpu.c
 CpuInit, 162
 CpuMemcpy, 163
 CpuMemSet, 163
 CpuStartUserProgram, 164
ARMCM0_S32K11/flash.c
 blockInfo, 239
 bootBlockInfo, 239
 FlashAddToBlock, 234
 FlashCommandSequence, 234
 FlashDone, 235

FlashErase, 235
 FlashEraseSectors, 236
 FlashGetSectorIdx, 236
 Flash GetUserProgBaseAddress, 236
 FlashInit, 237
 FlashInitBlock, 237
 FlashSwitchBlock, 237
 FlashVerifyChecksum, 238
 FlashWrite, 238
 FlashWriteBlock, 238
 FlashWriteChecksum, 239
ARMCM0_S32K11/flash.h
 FlashDone, 469
 FlashErase, 469
 Flash GetUserProgBaseAddress, 470
 FlashInit, 470
 FlashVerifyChecksum, 471
 FlashWrite, 471
 FlashWriteChecksum, 472
ARMCM0_S32K11/GCC/cpu_comp.c
 CpuIrqDisable, 574
 CpuIrqEnable, 574
ARMCM0_S32K11/IAR/cpu_comp.c
 CpuIrqDisable, 575
 CpuIrqEnable, 575
ARMCM0_S32K11/nvm.c
 NvmDone, 674
 NvmErase, 674
 Nvm GetUserProgBaseAddress, 674
 NvmInit, 675
 NvmVerifyChecksum, 675
 NvmWrite, 675
ARMCM0_S32K11/timer.c
 TimerGet, 767
 TimerInit, 768
 TimerReset, 768
 TimerUpdate, 768
ARMCM0_S32K11/types.h
 blt_addr, 833
 blt_bool, 833
 blt_char, 833
 blt_int16s, 834
 blt_int16u, 834
 blt_int32s, 834
 blt_int32u, 834
 blt_int8s, 834
 blt_int8u, 834
ARMCM0_STM32C0/cpu.c
 CpuInit, 165
 CpuMemcpy, 165
 CpuMemSet, 166
 CpuStartUserProgram, 166
ARMCM0_STM32C0/flash.c
 blockInfo, 248
 bootBlockInfo, 248
 FlashAddToBlock, 242
 FlashDone, 243
 FlashErase, 243
 FlashEraseSectors, 243
 FlashGetPage, 244
 FlashGetSector, 244
 FlashGetSectorBaseAddr, 244
 FlashGetSectorSize, 245
 Flash GetUserProgBaseAddress, 245
 FlashInit, 245
 FlashInitBlock, 246
 flashLayout, 249
 FlashSwitchBlock, 246
 FlashVerifyChecksum, 247
 FlashWrite, 247
 FlashWriteBlock, 247
 FlashWriteChecksum, 248
ARMCM0_STM32C0/flash.h
 FlashDone, 473
 FlashErase, 473
 Flash GetUserProgBaseAddress, 474
 FlashInit, 474
 FlashVerifyChecksum, 475
 FlashWrite, 475
 FlashWriteChecksum, 476
ARMCM0_STM32C0/GCC/cpu_comp.c
 CpuIrqDisable, 576
 CpuIrqEnable, 577
ARMCM0_STM32C0/IAR/cpu_comp.c
 CpuIrqDisable, 578
 CpuIrqEnable, 578
ARMCM0_STM32C0/Keil/cpu_comp.c
 CpuIrqDisable, 579
 CpuIrqEnable, 579
ARMCM0_STM32C0/nvm.c
 NvmDone, 676
 NvmErase, 677
 Nvm GetUserProgBaseAddress, 677
 NvmInit, 677
 NvmVerifyChecksum, 678
 NvmWrite, 678
ARMCM0_STM32C0/timer.c
 HAL_GetTick, 770
 TimerGet, 770
 TimerInit, 770
 TimerReset, 771
 TimerUpdate, 771
ARMCM0_STM32C0/types.h
 blt_addr, 835
 blt_bool, 835
 blt_char, 836
 blt_int16s, 836
 blt_int16u, 836
 blt_int32s, 836
 blt_int32u, 836
 blt_int64s, 836
 blt_int64u, 836
 blt_int8s, 837
 blt_int8u, 837
ARMCM0_STM32F0/can.c
 CanGetSpeedConfig, 92

CanInit, 93
CanReceivePacket, 93
canTiming, 94
CanTransmitPacket, 93
ARMCM0_STM32F0/cpu.c
 CpuInit, 167
 CpuMemcpy, 168
 CpuMemSet, 168
 CpuStartUserProgram, 169
ARMCM0_STM32F0/flash.c
 blockInfo, 257
 bootBlockInfo, 257
 FlashAddToBlock, 251
 FlashDone, 252
 FlashErase, 252
 FlashEraseSectors, 252
 FlashGetSector, 253
 FlashGetSectorBaseAddr, 253
 FlashGetSize, 254
 Flash GetUserProgBaseAddress, 254
 FlashInit, 254
 FlashInitBlock, 254
 flashLayout, 257
 FlashSwitchBlock, 255
 FlashVerifyChecksum, 255
 FlashWrite, 255
 FlashWriteBlock, 256
 FlashWriteChecksum, 256
ARMCM0_STM32F0/flash.h
 FlashDone, 477
 FlashErase, 477
 Flash GetUserProgBaseAddress, 478
 FlashInit, 478
 FlashVerifyChecksum, 479
 FlashWrite, 479
 FlashWriteChecksum, 480
ARMCM0_STM32F0/GCC/cpu_comp.c
 CpuIrqDisable, 580
 CpuIrqEnable, 580
ARMCM0_STM32F0/IAR/cpu_comp.c
 CpuIrqDisable, 581
 CpuIrqEnable, 582
ARMCM0_STM32F0/Keil/cpu_comp.c
 CpuIrqDisable, 583
 CpuIrqEnable, 583
ARMCM0_STM32F0/nvm.c
 NvmDone, 679
 NvmErase, 680
 Nvm GetUserProgBaseAddress, 680
 NvmInit, 680
 NvmVerifyChecksum, 681
 NvmWrite, 681
ARMCM0_STM32F0/timer.c
 HAL_GetTick, 773
 TimerGet, 773
 TimerInit, 773
 TimerReset, 773
 TimerUpdate, 774
ARMCM0_STM32F0/types.h
 blt_addr, 838
 blt_bool, 838
 blt_char, 838
 blt_int16s, 838
 blt_int16u, 838
 blt_int32s, 838
 blt_int32u, 839
 blt_int8s, 839
 blt_int8u, 839
ARMCM0_STM32G0/can.c
 CanGetSpeedConfig, 96
 CanInit, 96
 CanReceivePacket, 96
 canTiming, 97
 CanTransmitPacket, 97
ARMCM0_STM32G0/cpu.c
 CpuInit, 170
 CpuMemcpy, 170
 CpuMemSet, 171
 CpuStartUserProgram, 171
ARMCM0_STM32G0/flash.c
 blockInfo, 267
 bootBlockInfo, 268
 FlashAddToBlock, 260
 FlashDone, 261
 FlashErase, 261
 FlashEraseSectors, 261
 FlashGetBank, 262
 FlashGetPage, 262
 FlashGetSector, 263
 FlashGetSectorBaseAddr, 263
 FlashGetSize, 263
 Flash GetUserProgBaseAddress, 264
 FlashInit, 264
 FlashInitBlock, 264
 flashLayout, 268
 FlashSwitchBlock, 265
 FlashVerifyChecksum, 265
 FlashWrite, 265
 FlashWriteBlock, 267
 FlashWriteChecksum, 267
ARMCM0_STM32G0/flash.h
 FlashDone, 481
 FlashErase, 481
 Flash GetUserProgBaseAddress, 482
 FlashInit, 482
 FlashVerifyChecksum, 483
 FlashWrite, 483
 FlashWriteChecksum, 484
ARMCM0_STM32G0/GCC/cpu_comp.c
 CpuIrqDisable, 584
 CpuIrqEnable, 584
ARMCM0_STM32G0/IAR/cpu_comp.c
 CpuIrqDisable, 585
 CpuIrqEnable, 585
ARMCM0_STM32G0/Keil/cpu_comp.c
 CpuIrqDisable, 586

CpuIrqEnable, 587
 ARMCM0_STM32G0/nvm.c
 NvmDone, 682
 NvmErase, 683
 Nvm GetUserProgBaseAddress, 683
 NvmlInit, 683
 NvmVerifyChecksum, 684
 NvmWrite, 684
 ARMCM0_STM32G0/timer.c
 HAL_GetTick, 775
 TimerGet, 775
 TimerInit, 776
 TimerReset, 776
 TimerUpdate, 776
 ARMCM0_STM32G0/types.h
 blt_addr, 840
 blt_bool, 840
 blt_char, 840
 blt_int16s, 840
 blt_int16u, 841
 blt_int32s, 841
 blt_int32u, 841
 blt_int64s, 841
 blt_int64u, 841
 blt_int8s, 841
 blt_int8u, 841
 ARMCM0_STM32L0/cpu.c
 CpuInit, 172
 CpuMemcpy, 172
 CpuMemSet, 173
 CpuStartUserProgram, 173
 ARMCM0_STM32L0/flash.c
 blockInfo, 275
 bootBlockInfo, 276
 FlashAddToBlock, 270
 FlashDone, 271
 FlashErase, 271
 Flash GetUserProgBaseAddress, 272
 FlashInit, 272
 FlashInitBlock, 272
 flashLayout, 276
 FlashSwitchBlock, 273
 FlashVerifyChecksum, 273
 FlashWrite, 273
 FlashWriteBlock, 275
 FlashWriteChecksum, 275
 ARMCM0_STM32L0/flash.h
 FlashDone, 485
 FlashErase, 485
 Flash GetUserProgBaseAddress, 486
 FlashInit, 486
 FlashVerifyChecksum, 487
 FlashWrite, 487
 FlashWriteChecksum, 488
 ARMCM0_STM32L0/GCC/cpu_comp.c
 CpuIrqDisable, 588
 CpuIrqEnable, 588
 ARMCM0_STM32L0/IAR/cpu_comp.c
 CpuIrqDisable, 589
 CpuIrqEnable, 589
 ARMCM0_STM32L0/Keil/cpu_comp.c
 CpuIrqDisable, 590
 CpuIrqEnable, 590
 ARMCM0_STM32L0/nvm.c
 NvmDone, 685
 NvmErase, 686
 Nvm GetUserProgBaseAddress, 686
 NvmlInit, 686
 NvmVerifyChecksum, 687
 NvmWrite, 687
 ARMCM0_STM32L0/timer.c
 HAL_GetTick, 778
 TimerGet, 778
 TimerInit, 778
 TimerReset, 779
 TimerUpdate, 779
 ARMCM0_STM32L0/types.h
 blt_addr, 842
 blt_bool, 843
 blt_char, 843
 blt_int16s, 843
 blt_int16u, 843
 blt_int32s, 843
 blt_int32u, 843
 blt_int8s, 843
 blt_int8u, 844
 ARMCM0_XMC1/can.c
 CanInit, 99
 CanReceivePacket, 99
 CanTransmitPacket, 100
 ARMCM0_XMC1/cpu.c
 CpuInit, 174
 CpuMemcpy, 175
 CpuMemSet, 175
 CpuStartUserProgram, 176
 ARMCM0_XMC1/flash.c
 blockInfo, 284
 bootBlockInfo, 285
 FlashAddToBlock, 279
 FlashDone, 279
 FlashErase, 279
 FlashEraseSectors, 280
 FlashGetSector, 280
 FlashGetSectorBaseAddr, 280
 Flash GetUserProgBaseAddress, 281
 FlashInit, 281
 FlashInitBlock, 281
 flashLayout, 285
 FlashSwitchBlock, 282
 FlashVerifyChecksum, 282
 FlashWrite, 282
 FlashWriteBlock, 284
 FlashWriteChecksum, 284
 ARMCM0_XMC1/flash.h
 FlashDone, 489
 FlashErase, 489

Flash GetUserProgBaseAddress, 490
FlashInit, 490
FlashVerifyChecksum, 491
FlashWrite, 491
FlashWriteChecksum, 492
ARMCM0_XMC1/GCC/cpu_comp.c
 CpuIrqDisable, 591
 CpuIrqEnable, 592
ARMCM0_XMC1/IAR/cpu_comp.c
 CpuIrqDisable, 593
 CpuIrqEnable, 593
ARMCM0_XMC1/Keil/cpu_comp.c
 CpuIrqDisable, 594
 CpuIrqEnable, 594
ARMCM0_XMC1/nvm.c
 NvmDone, 688
 NvmErase, 689
 Nvm GetUserProgBaseAddress, 689
 NvmInit, 689
 NvmVerifyChecksum, 690
 NvmWrite, 690
ARMCM0_XMC1/timer.c
 TimerGet, 780
 TimerInit, 781
 TimerReset, 781
 TimerUpdate, 781
ARMCM0_XMC1/types.h
 blt_addr, 845
 blt_bool, 845
 blt_char, 845
 blt_int16s, 845
 blt_int16u, 845
 blt_int32s, 845
 blt_int32u, 845
 blt_int8s, 846
 blt_int8u, 846
ARMCM33_STM32H5/can.c
 CanGetSpeedConfig, 101
 CanInit, 102
 CanReceivePacket, 102
 canTiming, 103
 CanTransmitPacket, 102
ARMCM33_STM32H5/cpu.c
 CpuInit, 177
 CpuMemcpy, 177
 CpuMemSet, 178
 CpuStartUserProgram, 178
ARMCM33_STM32H5/flash.c
 blockInfo, 295
 bootBlockInfo, 296
 FlashAddToBlock, 288
 FlashDone, 288
 FlashEmptyCheckSector, 288
 FlashErase, 290
 FlashEraseSectors, 290
 FlashGetBank, 291
 FlashGetPage, 291
 FlashGetSectorIdx, 291
Flash GetUserProgBaseAddress, 292
FlashInit, 292
FlashInitBlock, 292
flashLayout, 296
FlashSwitchBlock, 293
FlashVerifyChecksum, 293
FlashWrite, 293
FlashWriteBlock, 295
FlashWriteChecksum, 295
ARMCM33_STM32H5/flash.h
 FlashDone, 493
 FlashErase, 493
 Flash GetUserProgBaseAddress, 494
 FlashInit, 494
 FlashVerifyChecksum, 495
 FlashWrite, 495
 FlashWriteChecksum, 496
ARMCM33_STM32H5/GCC/cpu_comp.c
 CpuIrqDisable, 595
 CpuIrqEnable, 595
ARMCM33_STM32H5/IAR/cpu_comp.c
 CpuIrqDisable, 596
 CpuIrqEnable, 597
ARMCM33_STM32H5/Keil/cpu_comp.c
 CpuIrqDisable, 598
 CpuIrqEnable, 598
ARMCM33_STM32H5/nvm.c
 NvmDone, 691
 NvmErase, 692
 Nvm GetUserProgBaseAddress, 692
 NvmInit, 692
 NvmVerifyChecksum, 693
 NvmWrite, 693
ARMCM33_STM32H5/timer.c
 HAL_GetTick, 783
 TimerGet, 783
 TimerInit, 783
 TimerReset, 784
 TimerUpdate, 784
ARMCM33_STM32H5/types.h
 blt_addr, 847
 blt_bool, 847
 blt_char, 847
 blt_int16s, 847
 blt_int16u, 847
 blt_int32s, 847
 blt_int32u, 848
 blt_int64s, 848
 blt_int64u, 848
 blt_int8s, 848
 blt_int8u, 848
ARMCM33_STM32H5/usb.c
 tud_suspend_cb, 895
 UsbFree, 896
 UsbInit, 896
 UsbReceiveByte, 896
 UsbReceivePacket, 897
 UsbTransmitPacket, 897

ARMCM33_STM32L5/can.c
 CanGetSpeedConfig, 105
 CanInit, 105
 CanReceivePacket, 105
 canTiming, 106
 CanTransmitPacket, 106
 ARMCM33_STM32L5/cpu.c
 CpuInit, 179
 CpuMemcpy, 179
 CpuMemSet, 180
 CpuStartUserProgram, 180
 ARMCM33_STM32L5/flash.c
 blockInfo, 305
 bootBlockInfo, 305
 FlashAddToBlock, 299
 FlashDone, 299
 FlashErase, 299
 FlashEraseSectors, 300
 FlashGetBank, 300
 FlashGetPage, 301
 FlashGetPageSize, 301
 FlashGetSectorIdx, 301
 Flash GetUserProgBaseAddress, 302
 FlashInit, 302
 FlashInitBlock, 302
 FlashIsDualBankMode, 303
 flashLayout, 305
 FlashSwitchBlock, 303
 FlashVerifyChecksum, 303
 FlashWrite, 304
 FlashWriteBlock, 304
 FlashWriteChecksum, 305
 ARMCM33_STM32L5/flash.h
 FlashDone, 497
 FlashErase, 497
 Flash GetUserProgBaseAddress, 498
 FlashInit, 498
 FlashVerifyChecksum, 499
 FlashWrite, 499
 FlashWriteChecksum, 500
 ARMCM33_STM32L5/GCC/cpu_comp.c
 CpuIrqDisable, 599
 CpuIrqEnable, 599
 ARMCM33_STM32L5/IAR/cpu_comp.c
 CpuIrqDisable, 600
 CpuIrqEnable, 600
 ARMCM33_STM32L5/Keil/cpu_comp.c
 CpuIrqDisable, 601
 CpuIrqEnable, 602
 ARMCM33_STM32L5/nvm.c
 NvmDone, 694
 NvmErase, 695
 Nvm GetUserProgBaseAddress, 695
 NvmInit, 695
 NvmVerifyChecksum, 696
 NvmWrite, 696
 ARMCM33_STM32L5/timer.c
 HAL_GetTick, 786
 TimerGet, 786
 TimerInit, 786
 TimerReset, 786
 TimerUpdate, 787
 ARMCM33_STM32L5/types.h
 blt_addr, 849
 blt_bool, 849
 blt_char, 849
 blt_int16s, 850
 blt_int16u, 850
 blt_int32s, 850
 blt_int32u, 850
 blt_int64s, 850
 blt_int64u, 850
 blt_int8s, 850
 blt_int8u, 851
 ARMCM33_STM32L5/usb.c
 UsbFifoMgrCreate, 900
 UsbFifoMgrInit, 900
 UsbFifoMgrRead, 900
 UsbFifoMgrScan, 901
 UsbFifoMgrWrite, 901
 UsbFree, 902
 UsbInit, 902
 UsbReceiveByte, 902
 UsbReceivePacket, 902
 UsbReceivePipeBulkOUT, 903
 UsbTransmitByte, 903
 UsbTransmitPacket, 903
 UsbTransmitPipeBulkIN, 904
 ARMCM33_STM32U5/can.c
 CanGetSpeedConfig, 108
 CanInit, 109
 CanReceivePacket, 109
 canTiming, 110
 CanTransmitPacket, 109
 ARMCM33_STM32U5/cpu.c
 CpuInit, 181
 CpuMemcpy, 182
 CpuMemSet, 182
 CpuStartUserProgram, 183
 ARMCM33_STM32U5/flash.c
 blockInfo, 314
 bootBlockInfo, 314
 FlashAddToBlock, 308
 FlashDone, 309
 FlashEmptyCheckSector, 309
 FlashErase, 309
 FlashEraseSectors, 310
 FlashGetBank, 310
 FlashGetPage, 310
 FlashGetSectorIdx, 311
 Flash GetUserProgBaseAddress, 311
 FlashInit, 311
 FlashInitBlock, 312
 flashLayout, 315
 FlashSwitchBlock, 312
 FlashVerifyChecksum, 313

FlashWrite, 313
FlashWriteBlock, 313
FlashWriteChecksum, 314
ARMCM33_STM32U5/flash.h
FlashDone, 501
FlashErase, 501
Flash GetUserProgBaseAddress, 502
FlashInit, 502
FlashVerifyChecksum, 503
FlashWrite, 503
FlashWriteChecksum, 504
ARMCM33_STM32U5/GCC/cpu_comp.c
CpuIrqDisable, 603
CpuIrqEnable, 603
ARMCM33_STM32U5/IAR/cpu_comp.c
CpuIrqDisable, 604
CpuIrqEnable, 604
ARMCM33_STM32U5/Keil/cpu_comp.c
CpuIrqDisable, 605
CpuIrqEnable, 605
ARMCM33_STM32U5/nvm.c
NvmDone, 697
NvmErase, 698
Nvm GetUserProgBaseAddress, 698
NvmInit, 698
NvmVerifyChecksum, 699
NvmWrite, 699
ARMCM33_STM32U5/timer.c
HAL_GetTick, 788
TimerGet, 788
TimerInit, 789
TimerReset, 789
TimerUpdate, 789
ARMCM33_STM32U5/types.h
blt_addr, 852
blt_bool, 852
blt_char, 852
blt_int16s, 852
blt_int16u, 852
blt_int32s, 852
blt_int32u, 852
blt_int64s, 853
blt_int64u, 853
blt_int8s, 853
blt_int8u, 853
ARMCM33_STM32U5/usb.c
UsbFifoMgrCreate, 906
UsbFifoMgrInit, 906
UsbFifoMgrRead, 907
UsbFifoMgrScan, 907
UsbFifoMgrWrite, 908
UsbFree, 908
UsbInit, 908
UsbReceiveByte, 908
UsbReceivePacket, 909
UsbReceivePipeBulkOUT, 909
UsbTransmitByte, 909
UsbTransmitPacket, 911
UsbTransmitPipeBulkIN, 911
ARMCM3_EFM32/cpu.c
CpuInit, 184
CpuMemcpy, 184
CpuMemSet, 185
CpuStartUserProgram, 185
ARMCM3_EFM32/flash.c
blockInfo, 323
bootBlockInfo, 323
FlashAddToBlock, 317
FlashCalcPageSize, 318
FlashDone, 318
FlashErase, 318
FlashEraseSectors, 319
FlashGetSector, 319
FlashGetSectorBaseAddr, 319
FlashGetSectorSize, 320
Flash GetUserProgBaseAddress, 320
FlashInit, 320
FlashInitBlock, 321
flashLayout, 324
FlashSwitchBlock, 321
FlashVerifyChecksum, 322
FlashWrite, 322
FlashWriteBlock, 322
FlashWriteChecksum, 323
ARMCM3_EFM32/flash.h
FlashDone, 505
FlashErase, 505
Flash GetUserProgBaseAddress, 506
FlashInit, 506
FlashVerifyChecksum, 507
FlashWrite, 507
FlashWriteChecksum, 508
ARMCM3_EFM32/GCC/cpu_comp.c
CpuIrqDisable, 606
CpuIrqEnable, 607
ARMCM3_EFM32/IAR/cpu_comp.c
CpuIrqDisable, 608
CpuIrqEnable, 608
ARMCM3_EFM32/nvm.c
NvmDone, 700
NvmErase, 701
Nvm GetUserProgBaseAddress, 701
NvmInit, 701
NvmVerifyChecksum, 702
NvmWrite, 702
ARMCM3_EFM32/timer.c
TimerGet, 791
TimerInit, 791
TimerReset, 791
TimerUpdate, 791
ARMCM3_EFM32/types.h
blt_addr, 854
blt_bool, 854
blt_char, 854
blt_int16s, 854
blt_int16u, 854

blt_int32s, 855
 blt_int32u, 855
 blt_int8s, 855
 blt_int8u, 855
ARMCM3_LM3S/can.c
 CanInit, 112
 CanReceivePacket, 112
 CanSetBittiming, 112
 CanTransmitPacket, 113
ARMCM3_LM3S/cpu.c
 CpuInit, 186
 CpuMemcpy, 186
 CpuMemSet, 187
 CpuStartUserProgram, 187
ARMCM3_LM3S/flash.c
 blockInfo, 332
 bootBlockInfo, 332
 FlashAddToBlock, 326
 FlashDone, 327
 FlashErase, 327
 FlashEraseSectors, 327
 FlashGetSector, 328
 FlashGetSectorBaseAddr, 328
 FlashGetSectorSize, 329
 Flash GetUserProgBaseAddress, 329
 FlashInit, 329
 FlashInitBlock, 329
 flashLayout, 332
 FlashSwitchBlock, 330
 FlashVerifyChecksum, 330
 FlashWrite, 330
 FlashWriteBlock, 331
 FlashWriteChecksum, 331
ARMCM3_LM3S/flash.h
 FlashDone, 509
 FlashErase, 509
 Flash GetUserProgBaseAddress, 510
 FlashInit, 510
 FlashVerifyChecksum, 511
 FlashWrite, 511
 FlashWriteChecksum, 512
ARMCM3_LM3S/GCC/cpu_comp.c
 CpuIrqDisable, 609
 CpuIrqEnable, 609
ARMCM3_LM3S/IAR/cpu_comp.c
 CpuIrqDisable, 610
 CpuIrqEnable, 610
ARMCM3_LM3S/nvm.c
 NvmDone, 703
 NvmErase, 704
 Nvm GetUserProgBaseAddress, 704
 NvmInit, 704
 NvmVerifyChecksum, 705
 NvmWrite, 705
ARMCM3_LM3S/timer.c
 TimerGet, 793
 TimerInit, 793
 TimerReset, 793
 TimerUpdate, 793
ARMCM3_LM3S/types.h
 blt_addr, 856
 blt_bool, 856
 blt_char, 856
 blt_int16s, 856
 blt_int16u, 856
 blt_int32s, 857
 blt_int32u, 857
 blt_int8s, 857
 blt_int8u, 857
ARMCM3_STM32F1/can.c
 CanGetSpeedConfig, 114
 CanInit, 115
 CanReceivePacket, 115
 canTiming, 116
 CanTransmitPacket, 115
ARMCM3_STM32F1/cpu.c
 CpuInit, 188
 CpuMemcpy, 189
 CpuMemSet, 189
 CpuStartUserProgram, 190
ARMCM3_STM32F1/flash.c
 blockInfo, 339
 bootBlockInfo, 339
 FlashAddToBlock, 335
 FlashDone, 335
 FlashErase, 336
 Flash GetUserProgBaseAddress, 336
 FlashInit, 336
 FlashInitBlock, 337
 flashLayout, 340
 FlashSwitchBlock, 337
 FlashVerifyChecksum, 338
 FlashWrite, 338
 FlashWriteBlock, 338
 FlashWriteChecksum, 339
ARMCM3_STM32F1/flash.h
 FlashDone, 513
 FlashErase, 513
 Flash GetUserProgBaseAddress, 514
 FlashInit, 514
 FlashVerifyChecksum, 515
 FlashWrite, 515
 FlashWriteChecksum, 516
ARMCM3_STM32F1/GCC/cpu_comp.c
 CpuIrqDisable, 611
 CpuIrqEnable, 612
ARMCM3_STM32F1/IAR/cpu_comp.c
 CpuIrqDisable, 613
 CpuIrqEnable, 613
ARMCM3_STM32F1/Keil/cpu_comp.c
 CpuIrqDisable, 614
 CpuIrqEnable, 614
ARMCM3_STM32F1/nvm.c
 NvmDone, 706
 NvmErase, 707
 Nvm GetUserProgBaseAddress, 707

NvmlInit, 707
NvmVerifyChecksum, 708
NvmWrite, 708
ARMCM3_STM32F1/timer.c
 HAL_GetTick, 795
 TimerGet, 795
 TimerInit, 795
 TimerReset, 796
 TimerUpdate, 796
ARMCM3_STM32F1/types.h
 blt_addr, 858
 blt_bool, 858
 blt_char, 858
 blt_int16s, 858
 blt_int16u, 858
 blt_int32s, 859
 blt_int32u, 859
 blt_int8s, 859
 blt_int8u, 859
ARMCM3_STM32F1/usb.c
 UsbFifoMgrCreate, 913
 UsbFifoMgrInit, 914
 UsbFifoMgrRead, 914
 UsbFifoMgrScan, 914
 UsbFifoMgrWrite, 915
 UsbFree, 915
 UsbInit, 915
 UsbReceiveByte, 916
 UsbReceivePacket, 916
 UsbReceivePipeBulkOUT, 916
 UsbTransmitByte, 917
 UsbTransmitPacket, 917
 UsbTransmitPipeBulkIN, 917
ARMCM3_STM32F2/can.c
 CanGetSpeedConfig, 118
 CanInit, 118
 CanReceivePacket, 118
 canTiming, 119
 CanTransmitPacket, 119
ARMCM3_STM32F2/cpu.c
 CpuInit, 191
 CpuMemcpy, 191
 CpuMemSet, 191
 CpuStartUserProgram, 192
ARMCM3_STM32F2/flash.c
 blockInfo, 347
 bootBlockInfo, 348
 FlashAddToBlock, 342
 FlashDone, 343
 FlashEmptyCheckSector, 343
 FlashErase, 343
 FlashEraseSectors, 344
 FlashGetSector, 344
 Flash GetUserProgBaseAddress, 344
 FlashInit, 345
 FlashInitBlock, 345
 flashLayout, 348
 FlashSwitchBlock, 345
ARMCM3_STM32F2/flash.h
 FlashVerifyChecksum, 346
 FlashWrite, 346
 FlashWriteBlock, 347
 FlashWriteChecksum, 347
ARMCM3_STM32F2/GCC/cpu_comp.c
 CpuLrqDisable, 615
 CpuLrqEnable, 615
ARMCM3_STM32F2/IAR/cpu_comp.c
 CpuLrqDisable, 616
 CpuLrqEnable, 617
ARMCM3_STM32F2/Keil/cpu_comp.c
 CpuLrqDisable, 618
 CpuLrqEnable, 618
ARMCM3_STM32F2/nvm.c
 NvmDone, 709
 NvmErase, 710
 Nvm GetUserProgBaseAddress, 710
 NvmlInit, 710
 NvmVerifyChecksum, 711
 NvmWrite, 711
ARMCM3_STM32F2/timer.c
 HAL_GetTick, 798
 TimerGet, 798
 TimerInit, 798
 TimerReset, 798
 TimerUpdate, 799
ARMCM3_STM32F2/types.h
 blt_addr, 860
 blt_bool, 860
 blt_char, 860
 blt_int16s, 860
 blt_int16u, 860
 blt_int32s, 861
 blt_int32u, 861
 blt_int8s, 861
 blt_int8u, 861
ARMCM3_STM32F2/usb.c
 UsbFifoMgrCreate, 920
 UsbFifoMgrInit, 920
 UsbFifoMgrRead, 920
 UsbFifoMgrScan, 921
 UsbFifoMgrWrite, 921
 UsbFree, 922
 UsbInit, 922
 UsbReceiveByte, 922
 UsbReceivePacket, 922
 UsbReceivePipeBulkOUT, 923
 UsbTransmitByte, 923
 UsbTransmitPacket, 923
 UsbTransmitPipeBulkIN, 924

ARMCM3_STM32L1/cpu.c
 CpuInit, 193
 CpuMemcpy, 193
 CpuMemSet, 194
 CpuStartUserProgram, 194
 ARMCM3_STM32L1/flash.c
 blockInfo, 355
 bootBlockInfo, 356
 FlashAddToBlock, 350
 FlashDone, 351
 FlashErase, 351
 Flash GetUserProgBaseAddress, 352
 FlashInit, 352
 FlashInitBlock, 352
 flashLayout, 356
 FlashSwitchBlock, 353
 FlashVerifyChecksum, 353
 FlashWrite, 353
 FlashWriteBlock, 355
 FlashWriteChecksum, 355
 ARMCM3_STM32L1/flash.h
 FlashDone, 521
 FlashErase, 521
 Flash GetUserProgBaseAddress, 522
 FlashInit, 522
 FlashVerifyChecksum, 523
 FlashWrite, 523
 FlashWriteChecksum, 524
 ARMCM3_STM32L1/GCC/cpu_comp.c
 CpuIrqDisable, 619
 CpuIrqEnable, 619
 ARMCM3_STM32L1/IAR/cpu_comp.c
 CpuIrqDisable, 620
 CpuIrqEnable, 620
 ARMCM3_STM32L1/Keil/cpu_comp.c
 CpuIrqDisable, 621
 CpuIrqEnable, 622
 ARMCM3_STM32L1/nvm.c
 NvmDone, 712
 NvmErase, 713
 Nvm GetUserProgBaseAddress, 713
 NvmInit, 713
 NvmVerifyChecksum, 714
 NvmWrite, 714
 ARMCM3_STM32L1/timer.c
 HAL_GetTick, 800
 TimerGet, 800
 TimerInit, 801
 TimerReset, 801
 TimerUpdate, 801
 ARMCM3_STM32L1/types.h
 blt_addr, 862
 blt_bool, 862
 blt_char, 862
 blt_int16s, 862
 blt_int16u, 862
 blt_int32s, 863
 blt_int32u, 863
 blt_int8s, 863
 blt_int8u, 863
 ARMCM4_S32K14/can.c
 CanDisabledModeEnter, 122
 CanDisabledModeExit, 122
 CanFreezeModeEnter, 122
 CanFreezeModeExit, 122
 CanGetSpeedConfig, 123
 CanInit, 123
 CanReceivePacket, 123
 canTiming, 124
 CanTransmitPacket, 124
 ARMCM4_S32K14/cpu.c
 CpuInit, 195
 CpuMemcpy, 196
 CpuMemSet, 196
 CpuStartUserProgram, 197
 ARMCM4_S32K14/flash.c
 blockInfo, 364
 bootBlockInfo, 364
 FlashAddToBlock, 359
 FlashCommandSequence, 359
 FlashDone, 360
 FlashErase, 360
 FlashEraseSectors, 360
 FlashGetSectorIdx, 361
 Flash GetUserProgBaseAddress, 361
 FlashInit, 361
 FlashInitBlock, 362
 FlashSwitchBlock, 362
 FlashVerifyChecksum, 363
 FlashWrite, 363
 FlashWriteBlock, 363
 FlashWriteChecksum, 364
 ARMCM4_S32K14/flash.h
 FlashDone, 525
 FlashErase, 525
 Flash GetUserProgBaseAddress, 526
 FlashInit, 526
 FlashVerifyChecksum, 527
 FlashWrite, 527
 FlashWriteChecksum, 528
 ARMCM4_S32K14/GCC/cpu_comp.c
 CpuIrqDisable, 623
 CpuIrqEnable, 623
 ARMCM4_S32K14/IAR/cpu_comp.c
 CpuIrqDisable, 624
 CpuIrqEnable, 624
 ARMCM4_S32K14/nvm.c
 NvmDone, 715
 NvmErase, 716
 Nvm GetUserProgBaseAddress, 716
 NvmInit, 716
 NvmVerifyChecksum, 717
 NvmWrite, 717
 ARMCM4_S32K14/timer.c
 TimerGet, 803
 TimerInit, 803

TimerReset, 803
TimerUpdate, 803
ARMCM4_S32K14/types.h
 blt_addr, 864
 blt_bool, 864
 blt_char, 864
 blt_int16s, 864
 blt_int16u, 864
 blt_int32s, 865
 blt_int32u, 865
 blt_int8s, 865
 blt_int8u, 865
ARMCM4_STM32F3/can.c
 CanGetSpeedConfig, 126
 CanInit, 127
 CanReceivePacket, 127
 canTiming, 128
 CanTransmitPacket, 127
ARMCM4_STM32F3/cpu.c
 CpuInit, 198
 CpuMemcpy, 198
 CpuMemSet, 198
 CpuStartUserProgram, 199
ARMCM4_STM32F3/flash.c
 blockInfo, 371
 bootBlockInfo, 371
 FlashAddToBlock, 367
 FlashDone, 367
 FlashErase, 368
 Flash GetUserProgBaseAddress, 368
 FlashInit, 368
 FlashInitBlock, 369
 flashLayout, 372
 FlashSwitchBlock, 369
 FlashVerifyChecksum, 370
 FlashWrite, 370
 FlashWriteBlock, 370
 FlashWriteChecksum, 371
ARMCM4_STM32F3/flash.h
 FlashDone, 529
 FlashErase, 529
 Flash GetUserProgBaseAddress, 530
 FlashInit, 530
 FlashVerifyChecksum, 531
 FlashWrite, 531
 FlashWriteChecksum, 532
ARMCM4_STM32F3/GCC/cpu_comp.c
 CpuIrqDisable, 625
 CpuIrqEnable, 625
ARMCM4_STM32F3/IAR/cpu_comp.c
 CpuIrqDisable, 626
 CpuIrqEnable, 627
ARMCM4_STM32F3/Keil/cpu_comp.c
 CpuIrqDisable, 628
 CpuIrqEnable, 628
ARMCM4_STM32F3/nvm.c
 NvmDone, 718
 NvmErase, 719
 Nvm GetUserProgBaseAddress, 719
 NvmInit, 719
 NvmVerifyChecksum, 720
 NvmWrite, 720
ARMCM4_STM32F3/timer.c
 HAL_GetTick, 805
 TimerGet, 805
 TimerInit, 805
 TimerReset, 806
 TimerUpdate, 806
ARMCM4_STM32F3/types.h
 blt_addr, 866
 blt_bool, 866
 blt_char, 866
 blt_int16s, 866
 blt_int16u, 866
 blt_int32s, 867
 blt_int32u, 867
 blt_int8s, 867
 blt_int8u, 867
ARMCM4_STM32F3/usb.c
 UsbFifoMgrCreate, 926
 UsbFifoMgrInit, 926
 UsbFifoMgrRead, 927
 UsbFifoMgrScan, 927
 UsbFifoMgrWrite, 928
 UsbFree, 928
 UsbInit, 928
 UsbReceiveByte, 928
 UsbReceivePacket, 929
 UsbReceivePipeBulkOUT, 929
 UsbTransmitByte, 929
 UsbTransmitPacket, 931
 UsbTransmitPipeBulkIN, 931
ARMCM4_STM32F4/can.c
 CanGetSpeedConfig, 130
 CanInit, 130
 CanReceivePacket, 130
 canTiming, 131
 CanTransmitPacket, 131
ARMCM4_STM32F4/cpu.c
 CpuInit, 200
 CpuMemcpy, 200
 CpuMemSet, 201
 CpuStartUserProgram, 201
ARMCM4_STM32F4/flash.c
 blockInfo, 379
 bootBlockInfo, 380
 FlashAddToBlock, 374
 FlashDone, 375
 FlashEmptyCheckSector, 375
 FlashErase, 375
 FlashEraseSectors, 376
 FlashGetSector, 376
 Flash GetUserProgBaseAddress, 376
 FlashInit, 377
 FlashInitBlock, 377
 flashLayout, 380

FlashSwitchBlock, 377
 FlashVerifyChecksum, 378
 FlashWrite, 378
 FlashWriteBlock, 379
 FlashWriteChecksum, 379
 ARMCM4_STM32F4/flash.h
 FlashDone, 533
 FlashErase, 533
 Flash GetUserProgBaseAddress, 534
 FlashInit, 534
 FlashVerifyChecksum, 535
 FlashWrite, 535
 FlashWriteChecksum, 536
 ARMCM4_STM32F4/GCC/cpu_comp.c
 CpuIrqDisable, 629
 CpuIrqEnable, 629
 ARMCM4_STM32F4/IAR/cpu_comp.c
 CpuIrqDisable, 630
 CpuIrqEnable, 630
 ARMCM4_STM32F4/Keil/cpu_comp.c
 CpuIrqDisable, 631
 CpuIrqEnable, 632
 ARMCM4_STM32F4/nvm.c
 NvmDone, 721
 NvmErase, 722
 Nvm GetUserProgBaseAddress, 722
 NvmInit, 722
 NvmVerifyChecksum, 723
 NvmWrite, 723
 ARMCM4_STM32F4/timer.c
 HAL_GetTick, 808
 TimerGet, 808
 TimerInit, 808
 TimerReset, 808
 TimerUpdate, 809
 ARMCM4_STM32F4/types.h
 blt_addr, 868
 blt_bool, 868
 blt_char, 868
 blt_int16s, 868
 blt_int16u, 868
 blt_int32s, 869
 blt_int32u, 869
 blt_int8s, 869
 blt_int8u, 869
 ARMCM4_STM32F4/usb.c
 UsbFifoMgrCreate, 933
 UsbFifoMgrInit, 934
 UsbFifoMgrRead, 934
 UsbFifoMgrScan, 934
 UsbFifoMgrWrite, 935
 UsbFree, 935
 UsbInit, 935
 UsbReceiveByte, 936
 UsbReceivePacket, 936
 UsbReceivePipeBulkOUT, 936
 UsbTransmitByte, 937
 UsbTransmitPacket, 937
 UsbTransmitPipeBulkIN, 937
 ARMCM4_STM32G4/can.c
 CanGetSpeedConfig, 133
 CanInit, 134
 CanReceivePacket, 134
 canTiming, 135
 CanTransmitPacket, 134
 ARMCM4_STM32G4/cpu.c
 CpuInit, 202
 CpuMemcpy, 203
 CpuMemSet, 203
 CpuStartUserProgram, 204
 ARMCM4_STM32G4/flash.c
 blockInfo, 388
 bootBlockInfo, 389
 FlashAddToBlock, 383
 FlashDone, 383
 FlashErase, 383
 FlashEraseSectors, 384
 FlashGetBank, 384
 FlashGetPage, 385
 FlashGetSectorIdx, 385
 Flash GetUserProgBaseAddress, 385
 FlashInit, 386
 FlashInitBlock, 386
 flashLayout, 389
 FlashSwitchBlock, 386
 FlashVerifyBankMode, 387
 FlashVerifyChecksum, 387
 FlashWrite, 387
 FlashWriteBlock, 388
 FlashWriteChecksum, 388
 ARMCM4_STM32G4/flash.h
 FlashDone, 537
 FlashErase, 537
 Flash GetUserProgBaseAddress, 538
 FlashInit, 538
 FlashVerifyChecksum, 539
 FlashWrite, 539
 FlashWriteChecksum, 540
 ARMCM4_STM32G4/GCC/cpu_comp.c
 CpuIrqDisable, 633
 CpuIrqEnable, 633
 ARMCM4_STM32G4/IAR/cpu_comp.c
 CpuIrqDisable, 634
 CpuIrqEnable, 634
 ARMCM4_STM32G4/Keil/cpu_comp.c
 CpuIrqDisable, 635
 CpuIrqEnable, 635
 ARMCM4_STM32G4/nvm.c
 NvmDone, 724
 NvmErase, 725
 Nvm GetUserProgBaseAddress, 725
 NvmInit, 725
 NvmVerifyChecksum, 726
 NvmWrite, 726
 ARMCM4_STM32G4/timer.c
 HAL_GetTick, 810

TimerGet, 810
TimerInit, 811
TimerReset, 811
TimerUpdate, 811
ARMCM4_STM32G4/types.h
 blt_addr, 870
 blt_bool, 870
 blt_char, 870
 blt_int16s, 870
 blt_int16u, 871
 blt_int32s, 871
 blt_int32u, 871
 blt_int64s, 871
 blt_int64u, 871
 blt_int8s, 871
 blt_int8u, 871
ARMCM4_STM32L4/can.c
 CanGetSpeedConfig, 137
 CanInit, 137
 CanReceivePacket, 137
 canTiming, 138
 CanTransmitPacket, 138
ARMCM4_STM32L4/cpu.c
 CpuInit, 205
 CpuMemcpy, 205
 CpuMemSet, 205
 CpuStartUserProgram, 206
ARMCM4_STM32L4/flash.c
 blockInfo, 396
 bootBlockInfo, 397
 FlashAddToBlock, 392
 FlashDone, 392
 FlashErase, 392
 FlashGetBank, 393
 FlashGetPage, 393
 Flash GetUserProgBaseAddress, 393
 FlashInit, 394
 FlashInitBlock, 394
 flashLayout, 397
 FlashSwitchBlock, 394
 FlashVerifyChecksum, 395
 FlashWrite, 395
 FlashWriteBlock, 396
 FlashWriteChecksum, 396
ARMCM4_STM32L4/flash.h
 FlashDone, 541
 FlashErase, 541
 Flash GetUserProgBaseAddress, 542
 FlashInit, 542
 FlashVerifyChecksum, 543
 FlashWrite, 543
 FlashWriteChecksum, 544
ARMCM4_STM32L4/GCC/cpu_comp.c
 CpuIrqDisable, 636
 CpuIrqEnable, 637
ARMCM4_STM32L4/IAR/cpu_comp.c
 CpuIrqDisable, 638
 CpuIrqEnable, 638
ARMCM4_STM32L4/Keil/cpu_comp.c
 CpuIrqDisable, 639
 CpuIrqEnable, 639
ARMCM4_STM32L4/nvm.c
 NvmDone, 727
 NvmErase, 728
 Nvm GetUserProgBaseAddress, 728
 NvmInit, 728
 NvmVerifyChecksum, 729
 NvmWrite, 729
ARMCM4_STM32L4/timer.c
 HAL_GetTick, 813
 TimerGet, 813
 TimerInit, 813
 TimerReset, 814
 TimerUpdate, 814
ARMCM4_STM32L4/types.h
 blt_addr, 872
 blt_bool, 873
 blt_char, 873
 blt_int16s, 873
 blt_int16u, 873
 blt_int32s, 873
 blt_int32u, 873
 blt_int8s, 873
 blt_int8u, 874
ARMCM4_STM32L4/usb.c
 UsbFifoMgrCreate, 940
 UsbFifoMgrInit, 940
 UsbFifoMgrRead, 940
 UsbFifoMgrScan, 941
 UsbFifoMgrWrite, 941
 UsbFree, 942
 UsbInit, 942
 UsbReceiveByte, 942
 UsbReceivePacket, 942
 UsbReceivePipeBulkOUT, 943
 UsbTransmitByte, 943
 UsbTransmitPacket, 943
 UsbTransmitPipeBulkIN, 944
ARMCM4_TM4C/cpu.c
 CpuInit, 207
 CpuMemcpy, 207
 CpuMemSet, 208
 CpuStartUserProgram, 208
ARMCM4_TM4C/flash.c
 blockInfo, 405
 bootBlockInfo, 406
 FlashAddToBlock, 400
 FlashDone, 400
 FlashErase, 400
 FlashEraseSectors, 401
 FlashGetSector, 401
 FlashGetSectorBaseAddr, 402
 FlashGetSectorSize, 402
 Flash GetUserProgBaseAddress, 402
 FlashInit, 403
 FlashInitBlock, 403

flashLayout, 406
 FlashSwitchBlock, 403
 FlashVerifyChecksum, 404
 FlashWrite, 404
 FlashWriteBlock, 405
 FlashWriteChecksum, 405
ARMCM4_TM4C/flash.h
 FlashDone, 545
 FlashErase, 545
 Flash GetUserProgBaseAddress, 546
 FlashInit, 546
 FlashVerifyChecksum, 547
 FlashWrite, 547
 FlashWriteChecksum, 548
ARMCM4_TM4C/IAR/cpu_comp.c
 CpuIrqDisable, 640
 CpuIrqEnable, 640
ARMCM4_TM4C/nvm.c
 NvmDone, 730
 NvmErase, 731
 Nvm GetUserProgBaseAddress, 731
 NvmInit, 731
 NvmVerifyChecksum, 732
 NvmWrite, 732
ARMCM4_TM4C/timer.c
 TimerGet, 815
 TimerInit, 816
 TimerReset, 816
 TimerUpdate, 816
ARMCM4_TM4C/types.h
 blt_addr, 875
 blt_bool, 875
 blt_char, 875
 blt_int16s, 875
 blt_int16u, 875
 blt_int32s, 875
 blt_int32u, 875
 blt_int8s, 876
 blt_int8u, 876
ARMCM4_TM4C/usb.c
 tud_suspend_cb, 945
 UsbFree, 945
 UsbInit, 946
 UsbReceiveByte, 946
 UsbReceivePacket, 946
 UsbTransmitPacket, 947
ARMCM4_XMC4/can.c
 CanInit, 140
 CanReceivePacket, 140
 CanTransmitPacket, 141
ARMCM4_XMC4/cpu.c
 CpuInit, 209
 CpuMemcpy, 210
 CpuMemSet, 210
 CpuStartUserProgram, 211
ARMCM4_XMC4/flash.c
 blockInfo, 416
 bootBlockInfo, 416
 FlashAddToBlock, 409
 FlashDone, 409
 FlashEmptyCheckSector, 409
 FlashErase, 411
 FlashEraseSectors, 411
 FlashGetSector, 412
 FlashGetSectorBaseAddr, 412
 Flash GetUserProgBaseAddress, 412
 FlashInit, 413
 FlashInitBlock, 413
 flashLayout, 416
 FlashSwitchBlock, 413
 FlashTranslateToNonCachedAddress, 414
 FlashVerifyChecksum, 414
 FlashWrite, 414
 FlashWriteBlock, 415
 FlashWriteChecksum, 415
ARMCM4_XMC4/flash.h
 FlashDone, 549
 FlashErase, 549
 Flash GetUserProgBaseAddress, 550
 FlashInit, 550
 FlashVerifyChecksum, 551
 FlashWrite, 551
 FlashWriteChecksum, 552
ARMCM4_XMC4/GCC/cpu_comp.c
 CpuIrqDisable, 641
 CpuIrqEnable, 642
ARMCM4_XMC4/IAR/cpu_comp.c
 CpuIrqDisable, 643
 CpuIrqEnable, 643
ARMCM4_XMC4/Keil/cpu_comp.c
 CpuIrqDisable, 644
 CpuIrqEnable, 644
ARMCM4_XMC4/nvm.c
 NvmDone, 733
 NvmErase, 734
 Nvm GetUserProgBaseAddress, 734
 NvmInit, 734
 NvmVerifyChecksum, 735
 NvmWrite, 735
ARMCM4_XMC4/timer.c
 TimerGet, 817
 TimerInit, 818
 TimerReset, 818
 TimerUpdate, 818
ARMCM4_XMC4/types.h
 blt_addr, 877
 blt_bool, 877
 blt_char, 877
 blt_int16s, 877
 blt_int16u, 877
 blt_int32s, 877
 blt_int32u, 878
 blt_int8s, 878
 blt_int8u, 878
ARMCM4_XMC4/usb.c
 tud_suspend_cb, 948

UsbFree, 948
 UsbInit, 949
 UsbReceiveByte, 949
 UsbReceivePacket, 949
 UsbTransmitPacket, 950
ARMCM7_STM32F7/can.c
 CanGetSpeedConfig, 142
 CanInit, 143
 CanReceivePacket, 143
 canTiming, 144
 CanTransmitPacket, 143
ARMCM7_STM32F7/cpu.c
 CpuInit, 212
 CpuMemcpy, 212
 CpuMemSet, 212
 CpuStartUserProgram, 213
ARMCM7_STM32F7/flash.c
 blockInfo, 425
 bootBlockInfo, 425
 FlashAddToBlock, 419
 FlashDone, 420
 FlashEmptyCheckSector, 420
 FlashErase, 420
 FlashEraseSectors, 421
 FlashGetSector, 421
 Flash GetUserProgBaseAddress, 422
 FlashInit, 422
 FlashInitBlock, 422
 FlashIsSingleBankMode, 423
 flashLayout, 425
 FlashSwitchBlock, 423
 FlashVerifyChecksum, 423
 FlashWrite, 424
 FlashWriteBlock, 424
 FlashWriteChecksum, 425
ARMCM7_STM32F7/flash.h
 FlashDone, 553
 FlashErase, 553
 Flash GetUserProgBaseAddress, 554
 FlashInit, 554
 FlashVerifyChecksum, 555
 FlashWrite, 555
 FlashWriteChecksum, 556
ARMCM7_STM32F7/GCC/cpu_comp.c
 CpuIrqDisable, 645
 CpuIrqEnable, 645
ARMCM7_STM32F7/IAR/cpu_comp.c
 CpuIrqDisable, 646
 CpuIrqEnable, 647
ARMCM7_STM32F7/Keil/cpu_comp.c
 CpuIrqDisable, 648
 CpuIrqEnable, 648
ARMCM7_STM32F7/nvm.c
 NvmDone, 736
 NvmErase, 737
 Nvm GetUserProgBaseAddress, 737
 NvmInit, 737
 NvmVerifyChecksum, 738
 NvmWrite, 738
ARMCM7_STM32F7/timer.c
 HAL_GetTick, 820
 TimerGet, 820
 TimerInit, 820
 TimerReset, 821
 TimerUpdate, 821
ARMCM7_STM32F7/types.h
 blt_addr, 879
 blt_bool, 879
 blt_char, 879
 blt_int16s, 879
 blt_int16u, 879
 blt_int32s, 880
 blt_int32u, 880
 blt_int8s, 880
 blt_int8u, 880
ARMCM7_STM32F7/usb.c
 UsbFifoMgrCreate, 952
 UsbFifoMgrInit, 952
 UsbFifoMgrRead, 953
 UsbFifoMgrScan, 953
 UsbFifoMgrWrite, 954
 UsbFree, 954
 UsbInit, 954
 UsbReceiveByte, 954
 UsbReceivePacket, 955
 UsbReceivePipeBulkOUT, 955
 UsbTransmitByte, 955
 UsbTransmitPacket, 957
 UsbTransmitPipeBulkIN, 957
ARMCM7_STM32H7/can.c
 CanGetSpeedConfig, 146
 CanInit, 146
 CanReceivePacket, 146
 canTiming, 147
 CanTransmitPacket, 147
ARMCM7_STM32H7/cpu.c
 CpuInit, 214
 CpuMemcpy, 214
 CpuMemSet, 215
 CpuStartUserProgram, 215
ARMCM7_STM32H7/flash.c
 blockInfo, 433
 bootBlockInfo, 433
 FlashAddToBlock, 428
 FlashDone, 428
 FlashEmptyCheckSector, 429
 FlashErase, 429
 FlashEraseSectors, 430
 FlashGetSectorIdx, 430
 Flash GetUserProgBaseAddress, 430
 FlashInit, 431
 FlashInitBlock, 431
 flashLayout, 434
 FlashSwitchBlock, 431
 FlashVerifyChecksum, 432
 FlashWrite, 432

FlashWriteBlock, 432
 FlashWriteChecksum, 433
 ARMCM7_STM32H7/flash.h
 FlashDone, 557
 FlashErase, 557
 Flash GetUserProgBaseAddress, 558
 FlashInit, 558
 FlashVerifyChecksum, 559
 FlashWrite, 559
 FlashWriteChecksum, 560
 ARMCM7_STM32H7/GCC/cpu_comp.c
 CpuIrqDisable, 649
 CpuIrqEnable, 649
 ARMCM7_STM32H7/IAR/cpu_comp.c
 CpuIrqDisable, 650
 CpuIrqEnable, 650
 ARMCM7_STM32H7/Keil/cpu_comp.c
 CpuIrqDisable, 651
 CpuIrqEnable, 652
 ARMCM7_STM32H7/nvm.c
 NvmDone, 739
 NvmErase, 740
 Nvm GetUserProgBaseAddress, 740
 NvmInit, 740
 NvmVerifyChecksum, 741
 NvmWrite, 741
 ARMCM7_STM32H7/timer.c
 HAL_GetTick, 823
 TimerGet, 823
 TimerInit, 823
 TimerReset, 823
 TimerUpdate, 824
 ARMCM7_STM32H7/types.h
 blt_addr, 881
 blt_bool, 881
 blt_char, 881
 blt_int16s, 881
 blt_int16u, 881
 blt_int32s, 882
 blt_int32u, 882
 blt_int8s, 882
 blt_int8u, 882
 ARMCM7_STM32H7/usb.c
 UsbFifoMgrCreate, 959
 UsbFifoMgrInit, 960
 UsbFifoMgrRead, 960
 UsbFifoMgrScan, 960
 UsbFifoMgrWrite, 961
 UsbFree, 961
 UsbInit, 961
 UsbReceiveByte, 962
 UsbReceivePacket, 962
 UsbReceivePipeBulkOUT, 962
 UsbTransmitByte, 963
 UsbTransmitPacket, 963
 UsbTransmitPipeBulkIN, 963
 AssertFailure
 asserts.c, 964
 asserts.h, 965
 asserts.c, 964
 AssertFailure, 964
 asserts.h, 965
 AssertFailure, 965
 backdoor.c, 966
 BackDoorCheck, 966
 BackDoorInit, 967
 backdoor.h, 967
 BackDoorCheck, 968
 BackDoorInit, 968
 BackDoorCheck
 backdoor.c, 966
 backdoor.h, 968
 BackDoorInit
 backdoor.c, 967
 backdoor.h, 968
 bank_num
 tFlashSector, 73
 blockInfo
 _template/flash.c, 230
 ARMCM0_S32K11/flash.c, 239
 ARMCM0_STM32C0/flash.c, 248
 ARMCM0_STM32F0/flash.c, 257
 ARMCM0_STM32G0/flash.c, 267
 ARMCM0_STM32L0/flash.c, 275
 ARMCM0_XMC1/flash.c, 284
 ARMCM33_STM32H5/flash.c, 295
 ARMCM33_STM32L5/flash.c, 305
 ARMCM33_STM32U5/flash.c, 314
 ARMCM3_EFM32/flash.c, 323
 ARMCM3_LM3S/flash.c, 332
 ARMCM3_STM32F1/flash.c, 339
 ARMCM3_STM32F2/flash.c, 347
 ARMCM3_STM32L1/flash.c, 355
 ARMCM4_S32K14/flash.c, 364
 ARMCM4_STM32F3/flash.c, 371
 ARMCM4_STM32F4/flash.c, 379
 ARMCM4_STM32G4/flash.c, 388
 ARMCM4_STM32L4/flash.c, 396
 ARMCM4_TM4C/flash.c, 405
 ARMCM4_XMC4/flash.c, 416
 ARMCM7_STM32F7/flash.c, 425
 ARMCM7_STM32H7/flash.c, 433
 flash_ecc.c, 1006
 HCS12/flash.c, 443
 TRICORE_TC2/flash.c, 454
 TRICORE_TC3/flash.c, 464
 blt_addr
 _template/types.h, 831
 ARMCM0_S32K11/types.h, 833
 ARMCM0_STM32C0/types.h, 835
 ARMCM0_STM32F0/types.h, 838
 ARMCM0_STM32G0/types.h, 840
 ARMCM0_STM32L0/types.h, 842
 ARMCM0_XMC1/types.h, 845
 ARMCM33_STM32H5/types.h, 847
 ARMCM33_STM32L5/types.h, 849

ARMCM3_STM32U5/types.h, 852
ARMCM3_EFM32/types.h, 854
ARMCM3_LM3S/types.h, 856
ARMCM3_STM32F1/types.h, 858
ARMCM3_STM32F2/types.h, 860
ARMCM3_STM32L1/types.h, 862
ARMCM4_S32K14/types.h, 864
ARMCM4_STM32F3/types.h, 866
ARMCM4_STM32F4/types.h, 868
ARMCM4_STM32G4/types.h, 870
ARMCM4_STM32L4/types.h, 872
ARMCM4_TM4C/types.h, 875
ARMCM4_XMC4/types.h, 877
ARMCM7_STM32F7/types.h, 879
ARMCM7_STM32H7/types.h, 881
HCS12/types.h, 883
TRICORE_TC2/types.h, 885
TRICORE_TC3/types.h, 887

blt_bool
 _template/types.h, 831
 ARMCM0_S32K11/types.h, 833
 ARMCM0_STM32C0/types.h, 835
 ARMCM0_STM32F0/types.h, 838
 ARMCM0_STM32G0/types.h, 840
 ARMCM0_STM32L0/types.h, 843
 ARMCM0_XMC1/types.h, 845
 ARMCM33_STM32H5/types.h, 847
 ARMCM33_STM32L5/types.h, 849
 ARMCM33_STM32U5/types.h, 852
 ARMCM3_EFM32/types.h, 854
 ARMCM3_LM3S/types.h, 856
 ARMCM3_STM32F1/types.h, 858
 ARMCM3_STM32F2/types.h, 860
 ARMCM3_STM32L1/types.h, 862
 ARMCM4_S32K14/types.h, 864
 ARMCM4_STM32F3/types.h, 866
 ARMCM4_STM32F4/types.h, 868
 ARMCM4_STM32G4/types.h, 870
 ARMCM4_STM32L4/types.h, 873
 ARMCM4_TM4C/types.h, 875
 ARMCM4_XMC4/types.h, 877
 ARMCM7_STM32F7/types.h, 879
 ARMCM7_STM32H7/types.h, 881
 HCS12/types.h, 883
 TRICORE_TC2/types.h, 885
 TRICORE_TC3/types.h, 887

blt_char
 _template/types.h, 831
 ARMCM0_S32K11/types.h, 833
 ARMCM0_STM32C0/types.h, 836
 ARMCM0_STM32F0/types.h, 838
 ARMCM0_STM32G0/types.h, 840
 ARMCM0_STM32L0/types.h, 843
 ARMCM0_XMC1/types.h, 845
 ARMCM33_STM32H5/types.h, 847
 ARMCM33_STM32L5/types.h, 849
 ARMCM33_STM32U5/types.h, 852
 ARMCM3_EFM32/types.h, 854

blt_int16s
 _template/types.h, 832
 ARMCM0_S32K11/types.h, 834
 ARMCM0_STM32C0/types.h, 836
 ARMCM0_STM32F0/types.h, 838
 ARMCM0_STM32G0/types.h, 840
 ARMCM0_STM32L0/types.h, 843
 ARMCM0_XMC1/types.h, 845
 ARMCM33_STM32H5/types.h, 847
 ARMCM33_STM32L5/types.h, 850
 ARMCM33_STM32U5/types.h, 852
 ARMCM3_EFM32/types.h, 854
 ARMCM3_LM3S/types.h, 856
 ARMCM3_STM32F1/types.h, 858
 ARMCM3_STM32F2/types.h, 860
 ARMCM3_STM32L1/types.h, 862
 ARMCM4_S32K14/types.h, 864
 ARMCM4_STM32F3/types.h, 866
 ARMCM4_STM32F4/types.h, 868
 ARMCM4_STM32G4/types.h, 870
 ARMCM4_STM32L4/types.h, 873
 ARMCM4_TM4C/types.h, 875
 ARMCM4_XMC4/types.h, 877
 ARMCM7_STM32F7/types.h, 879
 ARMCM7_STM32H7/types.h, 881
 HCS12/types.h, 883
 TRICORE_TC2/types.h, 885
 TRICORE_TC3/types.h, 887

blt_int16u
 _template/types.h, 832
 ARMCM0_S32K11/types.h, 834
 ARMCM0_STM32C0/types.h, 836
 ARMCM0_STM32F0/types.h, 838
 ARMCM0_STM32G0/types.h, 841
 ARMCM0_STM32L0/types.h, 843
 ARMCM0_XMC1/types.h, 845
 ARMCM33_STM32H5/types.h, 847
 ARMCM33_STM32L5/types.h, 850
 ARMCM33_STM32U5/types.h, 852
 ARMCM3_EFM32/types.h, 854
 ARMCM3_LM3S/types.h, 856
 ARMCM3_STM32F1/types.h, 858

ARMCM3_STM32F2/types.h, 860
 ARMCM3_STM32L1/types.h, 862
 ARMCM4_S32K14/types.h, 864
 ARMCM4_STM32F3/types.h, 866
 ARMCM4_STM32F4/types.h, 868
 ARMCM4_STM32G4/types.h, 871
 ARMCM4_STM32L4/types.h, 873
 ARMCM4_TM4C/types.h, 875
 ARMCM4_XMC4/types.h, 877
 ARMCM7_STM32F7/types.h, 879
 ARMCM7_STM32H7/types.h, 881
 HCS12/types.h, 883
 TRICORE_TC2/types.h, 885
 TRICORE_TC3/types.h, 887
blt_int32s
 _template/types.h, 832
 ARMCM0_S32K11/types.h, 834
 ARMCM0_STM32C0/types.h, 836
 ARMCM0_STM32F0/types.h, 838
 ARMCM0_STM32G0/types.h, 841
 ARMCM0_STM32L0/types.h, 843
 ARMCM0_XMC1/types.h, 845
 ARMCM33_STM32H5/types.h, 847
 ARMCM33_STM32L5/types.h, 850
 ARMCM33_STM32U5/types.h, 852
 ARMCM3_EFM32/types.h, 855
 ARMCM3_LM3S/types.h, 857
 ARMCM3_STM32F1/types.h, 859
 ARMCM3_STM32F2/types.h, 861
 ARMCM3_STM32L1/types.h, 863
 ARMCM4_S32K14/types.h, 865
 ARMCM4_STM32F3/types.h, 867
 ARMCM4_STM32F4/types.h, 869
 ARMCM4_STM32G4/types.h, 871
 ARMCM4_STM32L4/types.h, 873
 ARMCM4_TM4C/types.h, 875
 ARMCM4_XMC4/types.h, 877
 ARMCM7_STM32F7/types.h, 880
 ARMCM7_STM32H7/types.h, 882
 HCS12/types.h, 884
 TRICORE_TC2/types.h, 886
 TRICORE_TC3/types.h, 888
blt_int32u
 _template/types.h, 832
 ARMCM0_S32K11/types.h, 834
 ARMCM0_STM32C0/types.h, 836
 ARMCM0_STM32F0/types.h, 839
 ARMCM0_STM32G0/types.h, 841
 ARMCM0_STM32L0/types.h, 843
 ARMCM0_XMC1/types.h, 845
 ARMCM33_STM32H5/types.h, 848
 ARMCM33_STM32L5/types.h, 850
 ARMCM33_STM32U5/types.h, 852
 ARMCM3_EFM32/types.h, 855
 ARMCM3_LM3S/types.h, 857
 ARMCM3_STM32F1/types.h, 859
 ARMCM3_STM32F2/types.h, 861
 ARMCM3_STM32L1/types.h, 863
 ARMCM4_S32K14/types.h, 865
 ARMCM4_STM32F3/types.h, 867
 ARMCM4_STM32F4/types.h, 869
 ARMCM4_STM32G4/types.h, 871
 ARMCM4_STM32L4/types.h, 873
 ARMCM4_TM4C/types.h, 876
 ARMCM4_XMC4/types.h, 878
 ARMCM7_STM32F7/types.h, 880
 ARMCM7_STM32H7/types.h, 882
 HCS12/types.h, 884
 TRICORE_TC2/types.h, 886
 TRICORE_TC3/types.h, 888
blt_int8s
 _template/types.h, 832
 ARMCM0_S32K11/types.h, 834
 ARMCM0_STM32C0/types.h, 837
 ARMCM0_STM32F0/types.h, 839
 ARMCM0_STM32G0/types.h, 841
 ARMCM0_STM32L0/types.h, 843
 ARMCM0_XMC1/types.h, 846
 ARMCM33_STM32H5/types.h, 848
 ARMCM33_STM32L5/types.h, 850
 ARMCM33_STM32U5/types.h, 853
 ARMCM3_EFM32/types.h, 855
 ARMCM3_LM3S/types.h, 857
 ARMCM3_STM32F1/types.h, 859
 ARMCM3_STM32F2/types.h, 861
 ARMCM3_STM32L1/types.h, 863
 ARMCM4_S32K14/types.h, 865
 ARMCM4_STM32F3/types.h, 867
 ARMCM4_STM32F4/types.h, 869
 ARMCM4_STM32G4/types.h, 871
 ARMCM4_STM32L4/types.h, 873
 ARMCM4_TM4C/types.h, 876
 ARMCM4_XMC4/types.h, 878
 ARMCM7_STM32F7/types.h, 880
 ARMCM7_STM32H7/types.h, 882
 HCS12/types.h, 884
 TRICORE_TC2/types.h, 886
 TRICORE_TC3/types.h, 888
blt_int8u
 _template/types.h, 832
 ARMCM0_S32K11/types.h, 834
 ARMCM0_STM32C0/types.h, 837

ARMCM0_STM32F0/types.h, 839
ARMCM0_STM32G0/types.h, 841
ARMCM0_STM32L0/types.h, 844
ARMCM0_XMC1/types.h, 846
ARMCM33_STM32H5/types.h, 848
ARMCM33_STM32L5/types.h, 851
ARMCM33_STM32U5/types.h, 853
ARMCM3_EFM32/types.h, 855
ARMCM3_LM3S/types.h, 857
ARMCM3_STM32F1/types.h, 859
ARMCM3_STM32F2/types.h, 861
ARMCM3_STM32L1/types.h, 863
ARMCM4_S32K14/types.h, 865
ARMCM4_STM32F3/types.h, 867
ARMCM4_STM32F4/types.h, 869
ARMCM4_STM32G4/types.h, 871
ARMCM4_STM32L4/types.h, 874
ARMCM4_TM4C/types.h, 876
ARMCM4_XMC4/types.h, 878
ARMCM7_STM32F7/types.h, 880
ARMCM7_STM32H7/types.h, 882
HCS12/types.h, 884
TRICORE_TC2/types.h, 886
TRICORE_TC3/types.h, 888
boot.c, 969
 BootInit, 969
 BootTask, 969
boot.h, 970
 BootInit, 971
 BootTask, 971
bootBlockInfo
 _template/flash.c, 231
 ARMCM0_S32K11/flash.c, 239
 ARMCM0_STM32C0/flash.c, 248
 ARMCM0_STM32F0/flash.c, 257
 ARMCM0_STM32G0/flash.c, 268
 ARMCM0_STM32L0/flash.c, 276
 ARMCM0_XMC1/flash.c, 285
 ARMCM33_STM32H5/flash.c, 296
 ARMCM33_STM32L5/flash.c, 305
 ARMCM33_STM32U5/flash.c, 314
 ARMCM3_EFM32/flash.c, 323
 ARMCM3_LM3S/flash.c, 332
 ARMCM3_STM32F1/flash.c, 339
 ARMCM3_STM32F2/flash.c, 348
 ARMCM3_STM32L1/flash.c, 356
 ARMCM4_S32K14/flash.c, 364
 ARMCM4_STM32F3/flash.c, 371
 ARMCM4_STM32F4/flash.c, 380
 ARMCM4_STM32G4/flash.c, 389
 ARMCM4_STM32L4/flash.c, 397
 ARMCM4_TM4C/flash.c, 406
 ARMCM4_XMC4/flash.c, 416
 ARMCM7_STM32F7/flash.c, 425
 ARMCM7_STM32H7/flash.c, 433
 flash_ecc.c, 1006
 HCS12/flash.c, 443
 TRICORE_TC2/flash.c, 454
 TRICORE_TC3/flash.c, 464
BootInit
 boot.c, 969
 boot.h, 971
Bootloader Core, 44
Bootloader Ports, 47
BootTask
 boot.c, 969
 boot.h, 971
CAN driver of a port, 21
can.c, 83, 86, 91, 95, 98, 100, 104, 107, 111, 113, 117, 120, 125, 129, 132, 136, 139, 141, 145, 148, 152, 156
can.h, 971
 CanInit, 972
 CanReceivePacket, 972
 CanTransmitPacket, 973
CanDisabledModeEnter
 ARMCM0_S32K11/can.c, 88
 ARMCM4_S32K14/can.c, 122
CanDisabledModeExit
 ARMCM0_S32K11/can.c, 88
 ARMCM4_S32K14/can.c, 122
CanFreezeModeEnter
 ARMCM0_S32K11/can.c, 88
 ARMCM4_S32K14/can.c, 122
CanFreezeModeExit
 ARMCM0_S32K11/can.c, 88
 ARMCM4_S32K14/can.c, 122
CanGetSpeedConfig
 _template/can.c, 84
 ARMCM0_S32K11/can.c, 89
 ARMCM0_STM32F0/can.c, 92
 ARMCM0_STM32G0/can.c, 96
 ARMCM33_STM32H5/can.c, 101
 ARMCM33_STM32L5/can.c, 105
 ARMCM33_STM32U5/can.c, 108
 ARMCM3_STM32F1/can.c, 114
 ARMCM3_STM32F2/can.c, 118
 ARMCM4_S32K14/can.c, 123
 ARMCM4_STM32F3/can.c, 126
 ARMCM4_STM32F4/can.c, 130
 ARMCM4_STM32G4/can.c, 133
 ARMCM4_STM32L4/can.c, 137
 ARMCM7_STM32F7/can.c, 142
 ARMCM7_STM32H7/can.c, 146
 HCS12/can.c, 150
 TRICORE_TC2/can.c, 153
 TRICORE_TC3/can.c, 157
CanInit
 _template/can.c, 84
 ARMCM0_S32K11/can.c, 89
 ARMCM0_STM32F0/can.c, 93
 ARMCM0_STM32G0/can.c, 96
 ARMCM0_XMC1/can.c, 99
 ARMCM33_STM32H5/can.c, 102
 ARMCM33_STM32L5/can.c, 105
 ARMCM33_STM32U5/can.c, 109

ARMCM3_LM3S/can.c, 112
 ARMCM3_STM32F1/can.c, 115
 ARMCM3_STM32F2/can.c, 118
 ARMCM4_S32K14/can.c, 123
 ARMCM4_STM32F3/can.c, 127
 ARMCM4_STM32F4/can.c, 130
 ARMCM4_STM32G4/can.c, 134
 ARMCM4_STM32L4/can.c, 137
 ARMCM4_XMC4/can.c, 140
 ARMCMT7_STM32F7/can.c, 143
 ARMCMT7_STM32H7/can.c, 146
 can.h, 972
 HCS12/can.c, 150
 TRICORE_TC2/can.c, 154
 TRICORE_TC3/can.c, 157
 CanReceivePacket
 _template/can.c, 85
 ARMCMO_S32K11/can.c, 89
 ARMCMO_STM32F0/can.c, 93
 ARMCMO_STM32G0/can.c, 96
 ARMCMO_XMC1/can.c, 99
 ARMCM33_STM32H5/can.c, 102
 ARMCM33_STM32L5/can.c, 105
 ARMCM33_STM32U5/can.c, 109
 ARMCM3_LM3S/can.c, 112
 ARMCM3_STM32F1/can.c, 115
 ARMCM3_STM32F2/can.c, 118
 ARMCM4_S32K14/can.c, 123
 ARMCM4_STM32F3/can.c, 127
 ARMCM4_STM32F4/can.c, 130
 ARMCM4_STM32G4/can.c, 134
 ARMCM4_STM32L4/can.c, 137
 ARMCM4_XMC4/can.c, 140
 ARMCMT7_STM32F7/can.c, 143
 ARMCMT7_STM32H7/can.c, 146
 can.h, 972
 HCS12/can.c, 150
 TRICORE_TC2/can.c, 154
 TRICORE_TC3/can.c, 158
 CanSetBittiming
 ARMCM3_LM3S/can.c, 112
 canTiming
 _template/can.c, 85
 ARMCMO_S32K11/can.c, 90
 ARMCMO_STM32F0/can.c, 94
 ARMCMO_STM32G0/can.c, 97
 ARMCM33_STM32H5/can.c, 103
 ARMCM33_STM32L5/can.c, 106
 ARMCM33_STM32U5/can.c, 110
 ARMCM3_STM32F1/can.c, 116
 ARMCM3_STM32F2/can.c, 119
 ARMCM4_S32K14/can.c, 124
 ARMCM4_STM32F3/can.c, 128
 ARMCM4_STM32F4/can.c, 131
 ARMCM4_STM32G4/can.c, 135
 ARMCM4_STM32L4/can.c, 138
 ARMCMT7_STM32F7/can.c, 144
 ARMCMT7_STM32H7/can.c, 147
 HCS12/can.c, 151
 TRICORE_TC2/can.c, 155
 TRICORE_TC3/can.c, 159
 CanTransmitPacket
 _template/can.c, 85
 ARMCMO_S32K11/can.c, 90
 ARMCMO_STM32F0/can.c, 93
 ARMCMO_STM32G0/can.c, 97
 ARMCMO_XMC1/can.c, 100
 ARMCM33_STM32H5/can.c, 102
 ARMCM33_STM32L5/can.c, 106
 ARMCM33_STM32U5/can.c, 109
 ARMCM3_LM3S/can.c, 113
 ARMCM3_STM32F1/can.c, 115
 ARMCM3_STM32F2/can.c, 119
 ARMCM4_S32K14/can.c, 124
 ARMCM4_STM32F3/can.c, 127
 ARMCM4_STM32F4/can.c, 131
 ARMCM4_STM32G4/can.c, 134
 ARMCM4_STM32L4/can.c, 138
 ARMCM4_XMC4/can.c, 141
 ARMCMT7_STM32F7/can.c, 143
 ARMCMT7_STM32H7/can.c, 147
 can.h, 973
 HCS12/can.c, 151
 TRICORE_TC2/can.c, 154
 TRICORE_TC3/can.c, 158
 cbtr0
 tCanRegs, 54
 cbtr1
 tCanRegs, 54
 cctl0
 tCanRegs, 55
 cctl1
 tCanRegs, 55
 cforc
 tTimerRegs, 76
 cidac
 tCanRegs, 55
 cidar0
 tCanRegs, 55
 cidar1
 tCanRegs, 55
 cidar2
 tCanRegs, 55
 cidar3
 tCanRegs, 55
 cidar4
 tCanRegs, 56
 cidar5
 tCanRegs, 56
 cidar6
 tCanRegs, 56
 cidar7
 tCanRegs, 56
 cidmr0
 tCanRegs, 56
 cidmr1

tCanRegs, 56
cidmr2
 tCanRegs, 56
cidmr3
 tCanRegs, 57
cidmr4
 tCanRegs, 57
cidmr5
 tCanRegs, 57
cidmr6
 tCanRegs, 57
cidmr7
 tCanRegs, 57
com.c, 973
 ComFree, 974
 ComGetActiveInterfaceMaxRxLen, 974
 ComGetActiveInterfaceMaxTxLen, 975
 ComInit, 975
 ComIsConnected, 975
 ComTask, 975
 ComTransmitPacket, 976
com.h, 976
 COM_IF_CAN, 978
 COM_IF_MBRTU, 978
 COM_IF_NET, 978
 COM_IF_OTHER, 978
 COM_IF_RS232, 978
 COM_IF_USB, 978
 ComFree, 979
 ComGetActiveInterfaceMaxRxLen, 979
 ComGetActiveInterfaceMaxTxLen, 979
 ComInit, 979
 ComIsConnected, 980
 ComTask, 980
 ComTransmitPacket, 980
 tComInterfaceld, 978
COM_IF_CAN
 com.h, 978
COM_IF_MBRTU
 com.h, 978
COM_IF_NET
 com.h, 978
COM_IF_OTHER
 com.h, 978
COM_IF_RS232
 com.h, 978
COM_IF_USB
 com.h, 978
ComFree
 com.c, 974
 com.h, 979
ComGetActiveInterfaceMaxRxLen
 com.c, 974
 com.h, 979
ComGetActiveInterfaceMaxTxLen
 com.c, 975
 com.h, 979
ComInit
 com.c, 975
 com.h, 979
ComIsConnected
 com.c, 975
 com.h, 980
Compiler specifics of a port, 22
ComTask
 com.c, 975
 com.h, 980
ComTransmitPacket
 com.c, 976
 com.h, 980
connected
 tXcpInfo, 79
cop.c, 981
 CopInit, 981
 CopService, 982
cop.h, 982
 CopInit, 983
 CopService, 983
CopInit
 cop.c, 981
 cop.h, 983
CopService
 cop.c, 982
 cop.h, 983
CPU driver of a port, 21
cpu.c, 159, 162, 164, 166, 169, 171, 174, 176, 178, 181, 183, 185, 188, 190, 192, 195, 197, 199, 202, 204, 206, 209, 211, 213, 216, 218, 221
cpu.h, 983
 CpuInit, 984
 CpuIrqDisable, 984
 CpuIrqEnable, 984
 CpuMemcpy, 985
 CpuMemSet, 985
 CpuStartUserProgram, 986
cpu_comp.c, 572, 573, 575–578, 580–583, 585–588, 590–593, 595–598, 600–603, 605–608, 610–613, 615–618, 620–623, 625–628, 630–633, 635–638, 640–643, 645–648, 650–653, 655
CPU_USER_PROGRAM_STARTADDR_PTR
 HCS12/cpu.c, 216
CpuEnableUncorrectableBitErrorTrap
 TRICORE_TC2/cpu.c, 219
CpuEnableUncorrectableEccErrorTrap
 TRICORE_TC3/cpu.c, 222
CpuInit
 _template/cpu.c, 160
 ARMCM0_S32K11/cpu.c, 162
 ARMCM0_STM32C0/cpu.c, 165
 ARMCM0_STM32F0/cpu.c, 167
 ARMCM0_STM32G0/cpu.c, 170
 ARMCM0_STM32L0/cpu.c, 172
 ARMCM0_XMC1/cpu.c, 174
 ARMCM33_STM32H5/cpu.c, 177
 ARMCM33_STM32L5/cpu.c, 179
 ARMCM33_STM32U5/cpu.c, 181

- ARMCM3_EFM32/cpu.c, 184
 ARMCM3_LM3S/cpu.c, 186
 ARMCM3_STM32F1/cpu.c, 188
 ARMCM3_STM32F2/cpu.c, 191
 ARMCM3_STM32L1/cpu.c, 193
 ARMCM4_S32K14/cpu.c, 195
 ARMCM4_STM32F3/cpu.c, 198
 ARMCM4_STM32F4/cpu.c, 200
 ARMCM4_STM32G4/cpu.c, 202
 ARMCM4_STM32L4/cpu.c, 205
 ARMCM4_TM4C/cpu.c, 207
 ARMCM4_XMC4/cpu.c, 209
 ARMCM7_STM32F7/cpu.c, 212
 ARMCM7_STM32H7/cpu.c, 214
 cpu.h, 984
 HCS12/cpu.c, 217
 TRICORE_TC2/cpu.c, 220
 TRICORE_TC3/cpu.c, 222
- CpuIrqDisable
 _template/GCC/cpu_comp.c, 573
 ARMCM0_S32K11/GCC/cpu_comp.c, 574
 ARMCM0_S32K11/IAR/cpu_comp.c, 575
 ARMCM0_STM32C0/GCC/cpu_comp.c, 576
 ARMCM0_STM32C0/IAR/cpu_comp.c, 578
 ARMCM0_STM32C0/Keil/cpu_comp.c, 579
 ARMCM0_STM32F0/GCC/cpu_comp.c, 580
 ARMCM0_STM32F0/IAR/cpu_comp.c, 581
 ARMCM0_STM32F0/Keil/cpu_comp.c, 583
 ARMCM0_STM32G0/GCC/cpu_comp.c, 584
 ARMCM0_STM32G0/IAR/cpu_comp.c, 585
 ARMCM0_STM32G0/Keil/cpu_comp.c, 586
 ARMCM0_STM32L0/GCC/cpu_comp.c, 588
 ARMCM0_STM32L0/IAR/cpu_comp.c, 589
 ARMCM0_STM32L0/Keil/cpu_comp.c, 590
 ARMCM0_XMC1/GCC/cpu_comp.c, 591
 ARMCM0_XMC1/IAR/cpu_comp.c, 593
 ARMCM0_XMC1/Keil/cpu_comp.c, 594
 ARMCM33_STM32H5/GCC/cpu_comp.c, 595
 ARMCM33_STM32H5/IAR/cpu_comp.c, 596
 ARMCM33_STM32H5/Keil/cpu_comp.c, 598
 ARMCM33_STM32L5/GCC/cpu_comp.c, 599
 ARMCM33_STM32L5/IAR/cpu_comp.c, 600
 ARMCM33_STM32L5/Keil/cpu_comp.c, 601
 ARMCM33_STM32U5/GCC/cpu_comp.c, 603
 ARMCM33_STM32U5/IAR/cpu_comp.c, 604
 ARMCM33_STM32U5/Keil/cpu_comp.c, 605
 ARMCM3_EFM32/GCC/cpu_comp.c, 606
 ARMCM3_EFM32/IAR/cpu_comp.c, 608
 ARMCM3_LM3S/GCC/cpu_comp.c, 609
 ARMCM3_LM3S/IAR/cpu_comp.c, 610
 ARMCM3_STM32F1/GCC/cpu_comp.c, 611
 ARMCM3_STM32F1/IAR/cpu_comp.c, 613
 ARMCM3_STM32F1/Keil/cpu_comp.c, 614
 ARMCM3_STM32F2/GCC/cpu_comp.c, 615
 ARMCM3_STM32F2/IAR/cpu_comp.c, 616
 ARMCM3_STM32F2/Keil/cpu_comp.c, 618
 ARMCM3_STM32L1/GCC/cpu_comp.c, 619
 ARMCM3_STM32L1/IAR/cpu_comp.c, 620
- ARMCM3_STM32L1/Keil/cpu_comp.c, 621
 ARMCM4_S32K14/GCC/cpu_comp.c, 623
 ARMCM4_S32K14/IAR/cpu_comp.c, 624
 ARMCM4_STM32F3/GCC/cpu_comp.c, 625
 ARMCM4_STM32F3/IAR/cpu_comp.c, 626
 ARMCM4_STM32F3/Keil/cpu_comp.c, 628
 ARMCM4_STM32F4/GCC/cpu_comp.c, 629
 ARMCM4_STM32F4/IAR/cpu_comp.c, 630
 ARMCM4_STM32F4/Keil/cpu_comp.c, 631
 ARMCM4_STM32G4/GCC/cpu_comp.c, 633
 ARMCM4_STM32G4/IAR/cpu_comp.c, 634
 ARMCM4_STM32G4/Keil/cpu_comp.c, 635
 ARMCM4_STM32L4/GCC/cpu_comp.c, 636
 ARMCM4_STM32L4/IAR/cpu_comp.c, 638
 ARMCM4_STM32L4/Keil/cpu_comp.c, 639
 ARMCM4_TM4C/IAR/cpu_comp.c, 640
 ARMCM4_XMC4/GCC/cpu_comp.c, 641
 ARMCM4_XMC4/IAR/cpu_comp.c, 643
 ARMCM4_XMC4/Keil/cpu_comp.c, 644
 ARMCM7_STM32F7/GCC/cpu_comp.c, 645
 ARMCM7_STM32F7/IAR/cpu_comp.c, 646
 ARMCM7_STM32F7/Keil/cpu_comp.c, 648
 ARMCM7_STM32H7/GCC/cpu_comp.c, 649
 ARMCM7_STM32H7/IAR/cpu_comp.c, 650
 ARMCM7_STM32H7/Keil/cpu_comp.c, 651
 cpu.h, 984
 HCS12/CodeWarrior/cpu_comp.c, 653
 TRICORE_TC2/Tasking/cpu_comp.c, 654
 TRICORE_TC3/Tasking/cpu_comp.c, 655
- CpuIrqEnable
 _template/GCC/cpu_comp.c, 573
 ARMCM0_S32K11/GCC/cpu_comp.c, 574
 ARMCM0_S32K11/IAR/cpu_comp.c, 575
 ARMCM0_STM32C0/GCC/cpu_comp.c, 577
 ARMCM0_STM32C0/IAR/cpu_comp.c, 578
 ARMCM0_STM32C0/Keil/cpu_comp.c, 579
 ARMCM0_STM32F0/GCC/cpu_comp.c, 580
 ARMCM0_STM32F0/IAR/cpu_comp.c, 582
 ARMCM0_STM32F0/Keil/cpu_comp.c, 583
 ARMCM0_STM32G0/GCC/cpu_comp.c, 584
 ARMCM0_STM32G0/IAR/cpu_comp.c, 585
 ARMCM0_STM32G0/Keil/cpu_comp.c, 587
 ARMCM0_STM32L0/GCC/cpu_comp.c, 588
 ARMCM0_STM32L0/IAR/cpu_comp.c, 589
 ARMCM0_STM32L0/Keil/cpu_comp.c, 590
 ARMCM0_XMC1/GCC/cpu_comp.c, 592
 ARMCM0_XMC1/IAR/cpu_comp.c, 593
 ARMCM0_XMC1/Keil/cpu_comp.c, 594
 ARMCM33_STM32H5/GCC/cpu_comp.c, 595
 ARMCM33_STM32H5/IAR/cpu_comp.c, 597
 ARMCM33_STM32H5/Keil/cpu_comp.c, 598
 ARMCM33_STM32L5/GCC/cpu_comp.c, 599
 ARMCM33_STM32L5/IAR/cpu_comp.c, 600
 ARMCM33_STM32L5/Keil/cpu_comp.c, 602
 ARMCM33_STM32U5/GCC/cpu_comp.c, 603
 ARMCM33_STM32U5/IAR/cpu_comp.c, 604
 ARMCM33_STM32U5/Keil/cpu_comp.c, 605
 ARMCM3_EFM32/GCC/cpu_comp.c, 607

- ARMCM3_EFM32/IAR/cpu_comp.c, 608
ARMCM3_LM3S/GCC/cpu_comp.c, 609
ARMCM3_LM3S/IAR/cpu_comp.c, 610
ARMCM3_STM32F1/GCC/cpu_comp.c, 612
ARMCM3_STM32F1/IAR/cpu_comp.c, 613
ARMCM3_STM32F1/Keil/cpu_comp.c, 614
ARMCM3_STM32F2/GCC/cpu_comp.c, 615
ARMCM3_STM32F2/IAR/cpu_comp.c, 617
ARMCM3_STM32F2/Keil/cpu_comp.c, 618
ARMCM3_STM32L1/GCC/cpu_comp.c, 619
ARMCM3_STM32L1/IAR/cpu_comp.c, 620
ARMCM3_STM32L1/Keil/cpu_comp.c, 622
ARMCM4_S32K14/GCC/cpu_comp.c, 623
ARMCM4_S32K14/IAR/cpu_comp.c, 624
ARMCM4_STM32F3/GCC/cpu_comp.c, 625
ARMCM4_STM32F3/IAR/cpu_comp.c, 627
ARMCM4_STM32F3/Keil/cpu_comp.c, 628
ARMCM4_STM32F4/GCC/cpu_comp.c, 629
ARMCM4_STM32F4/IAR/cpu_comp.c, 630
ARMCM4_STM32F4/Keil/cpu_comp.c, 632
ARMCM4_STM32G4/GCC/cpu_comp.c, 633
ARMCM4_STM32G4/IAR/cpu_comp.c, 634
ARMCM4_STM32G4/Keil/cpu_comp.c, 635
ARMCM4_STM32L4/GCC/cpu_comp.c, 637
ARMCM4_STM32L4/IAR/cpu_comp.c, 638
ARMCM4_STM32L4/Keil/cpu_comp.c, 639
ARMCM4_TM4C/IAR/cpu_comp.c, 640
ARMCM4_XMC4/GCC/cpu_comp.c, 642
ARMCM4_XMC4/IAR/cpu_comp.c, 643
ARMCM4_XMC4/Keil/cpu_comp.c, 644
ARMCM7_STM32F7/GCC/cpu_comp.c, 645
ARMCM7_STM32F7/IAR/cpu_comp.c, 647
ARMCM7_STM32F7/Keil/cpu_comp.c, 648
ARMCM7_STM32H7/GCC/cpu_comp.c, 649
ARMCM7_STM32H7/IAR/cpu_comp.c, 650
ARMCM7_STM32H7/Keil/cpu_comp.c, 652
cpu.h, 984
HCS12/CodeWarrior/cpu_comp.c, 653
TRICORE_TC2/Tasking/cpu_comp.c, 654
TRICORE_TC3/Tasking/cpu_comp.c, 655
- CpuMemcpy
_template/cpu.c, 160
ARMCM0_S32K11/cpu.c, 163
ARMCM0_STM32C0/cpu.c, 165
ARMCM0_STM32F0/cpu.c, 168
ARMCM0_STM32G0/cpu.c, 170
ARMCM0_STM32L0/cpu.c, 172
ARMCM0_XMC1/cpu.c, 175
ARMCM33_STM32H5/cpu.c, 177
ARMCM33_STM32L5/cpu.c, 179
ARMCM33_STM32U5/cpu.c, 182
ARMCM3_EFM32/cpu.c, 184
ARMCM3_LM3S/cpu.c, 186
ARMCM3_STM32F1/cpu.c, 189
ARMCM3_STM32F2/cpu.c, 191
ARMCM3_STM32L1/cpu.c, 193
ARMCM4_S32K14/cpu.c, 196
ARMCM4_STM32F3/cpu.c, 198
- ARMCM4_STM32F4/cpu.c, 200
ARMCM4_STM32G4/cpu.c, 203
ARMCM4_STM32L4/cpu.c, 205
ARMCM4_TM4C/cpu.c, 207
ARMCM4_XMC4/cpu.c, 210
ARMCM7_STM32F7/cpu.c, 212
ARMCM7_STM32H7/cpu.c, 214
cpu.h, 985
HCS12/cpu.c, 217
TRICORE_TC2/cpu.c, 220
TRICORE_TC3/cpu.c, 222
- CpuMemSet
_template/cpu.c, 161
ARMCM0_S32K11/cpu.c, 163
ARMCM0_STM32C0/cpu.c, 166
ARMCM0_STM32F0/cpu.c, 168
ARMCM0_STM32G0/cpu.c, 171
ARMCM0_STM32L0/cpu.c, 173
ARMCM0_XMC1/cpu.c, 175
ARMCM33_STM32H5/cpu.c, 178
ARMCM33_STM32L5/cpu.c, 180
ARMCM33_STM32U5/cpu.c, 182
ARMCM3_EFM32/cpu.c, 185
ARMCM3_LM3S/cpu.c, 187
ARMCM3_STM32F1/cpu.c, 189
ARMCM3_STM32F2/cpu.c, 191
ARMCM3_STM32L1/cpu.c, 194
ARMCM4_S32K14/cpu.c, 196
ARMCM4_STM32F3/cpu.c, 198
ARMCM4_STM32F4/cpu.c, 201
ARMCM4_STM32G4/cpu.c, 203
ARMCM4_STM32L4/cpu.c, 205
ARMCM4_TM4C/cpu.c, 208
ARMCM4_XMC4/cpu.c, 210
ARMCM7_STM32F7/cpu.c, 212
ARMCM7_STM32H7/cpu.c, 215
cpu.h, 985
HCS12/cpu.c, 217
TRICORE_TC2/cpu.c, 220
TRICORE_TC3/cpu.c, 223
- CpuStartUserProgram
_template/cpu.c, 161
ARMCM0_S32K11/cpu.c, 164
ARMCM0_STM32C0/cpu.c, 166
ARMCM0_STM32F0/cpu.c, 169
ARMCM0_STM32G0/cpu.c, 171
ARMCM0_STM32L0/cpu.c, 173
ARMCM0_XMC1/cpu.c, 176
ARMCM33_STM32H5/cpu.c, 178
ARMCM33_STM32L5/cpu.c, 180
ARMCM33_STM32U5/cpu.c, 183
ARMCM3_EFM32/cpu.c, 185
ARMCM3_LM3S/cpu.c, 187
ARMCM3_STM32F1/cpu.c, 190
ARMCM3_STM32F2/cpu.c, 192
ARMCM3_STM32L1/cpu.c, 194
ARMCM4_S32K14/cpu.c, 197
ARMCM4_STM32F3/cpu.c, 199

ARMCM4_STM32F4/cpu.c, 201
 ARMCM4_STM32G4/cpu.c, 204
 ARMCM4_STM32L4/cpu.c, 206
 ARMCM4_TM4C/cpu.c, 208
 ARMCM4_XMC4/cpu.c, 211
 ARMCM7_STM32F7/cpu.c, 213
 ARMCM7_STM32H7/cpu.c, 215
 cpu.h, 986
 HCS12/cpu.c, 218
 TRICORE_TC2/cpu.c, 221
 TRICORE_TC3/cpu.c, 223
 crflg
 tCanRegs, 57
 crier
 tCanRegs, 57
 crxerr
 tCanRegs, 58
 ctaak
 tCanRegs, 58
 ctarq
 tCanRegs, 58
 ctbsel
 tCanRegs, 58
 ctflg
 tCanRegs, 58
 ctier
 tCanRegs, 58
 ctoData
 tXcpInfo, 79
 ctoLen
 tXcpInfo, 79
 ctoPending
 tXcpInfo, 80
 ctxerr
 tCanRegs, 58
 data
 tFifoPipe, 65
 tSrecLineParseObject, 75
 dfprot
 tFlashRegs, 69
 dlr
 tCanRxMsgSlot, 60
 tCanTxMsgSlot, 61
 dsr
 tCanRxMsgSlot, 60
 tCanTxMsgSlot, 61
 dummy
 tCanRxMsgSlot, 60
 dummy1
 tCanRegs, 59
 endptr
 tFifoCtrl, 63
 entries
 tFifoCtrl, 63
 fccob
 tFlashRegs, 69
 fccobix
 tFlashRegs, 69
 fclkdiv
 tFlashRegs, 70
 fcmd
 tFlashRegs, 70
 fcngf
 tFlashRegs, 70
 fercfg
 tFlashRegs, 70
 ferstat
 tFlashRegs, 70
 fifoctrlptr
 tFifoCtrl, 63
 file
 tFatFsObjects, 62
 file.c, 986
 FileHandleFirmwareUpdateRequest, 988
 FileInit, 988
 FileIsIdle, 989
 FileLibByteNibbleToChar, 989
 FileLibByteToHexString, 989
 FileLibHexStringToByte, 990
 FileLibLongToIntString, 990
 FileSrecGetLineType, 991
 FileSrecParseLine, 991
 FileSrecVerifyChecksum, 992
 FileTask, 992
 FIRMWARE_UPDATE_STATE_ERASING, 988
 FIRMWARE_UPDATE_STATE_IDLE, 988
 FIRMWARE_UPDATE_STATE_PROGRAMMING, 988
 FIRMWARE_UPDATE_STATE_STARTING, 988
 tFirmwareUpdateState, 988
 file.h, 992
 FileHandleFirmwareUpdateRequest, 995
 FileInit, 995
 FileIsIdle, 995
 FileSrecGetLineType, 995
 FileSrecParseLine, 996
 FileSrecVerifyChecksum, 996
 FileTask, 997
 LINE_TYPE_S1, 994
 LINE_TYPE_S2, 994
 LINE_TYPE_S3, 994
 LINE_TYPE_UNSUPPORTED, 994
 tSrecLineType, 994
 FileHandleFirmwareUpdateRequest
 file.c, 988
 file.h, 995
 FileInit
 file.c, 988
 file.h, 995
 FileIsIdle
 file.c, 989
 file.h, 995
 FileLibByteNibbleToChar
 file.c, 989

FileLibByteToHexString
 file.c, 989
FileLibHexStringToByte
 file.c, 990
FileLibLongToIntString
 file.c, 990
FileSrecGetLineType
 file.c, 991
 file.h, 995
FileSrecParseLine
 file.c, 991
 file.h, 996
FileSrecVerifyChecksum
 file.c, 992
 file.h, 996
FileTask
 file.c, 992
 file.h, 997
FIRMWARE_UPDATE_STATE_ERASING
 file.c, 988
FIRMWARE_UPDATE_STATE_IDLE
 file.c, 988
FIRMWARE_UPDATE_STATE_PROGRAMMING
 file.c, 988
FIRMWARE_UPDATE_STATE_STARTING
 file.c, 988
Flash driver of a port, 22
flash.c, 224, 232, 240, 249, 258, 269, 277, 286, 297,
 306, 315, 324, 333, 340, 349, 357, 365, 372,
 381, 390, 398, 407, 417, 426, 434, 445, 455
flash.h, 465, 468, 472, 476, 480, 484, 488, 492, 496,
 500, 504, 508, 512, 516, 520, 524, 528, 532,
 536, 540, 544, 548, 552, 556, 560, 564, 568
flash_ecc.c, 997
 blockInfo, 1006
 bootBlockInfo, 1006
 FlashAddToBlock, 1000
 FlashDone, 1001
 FlashErase, 1001
 flashExecCmd, 1007
 FlashExecuteCommand, 1001
 FlashGetGlobalAddrByte, 1002
 FlashGetPhysAddr, 1002
 FlashGetPhysPage, 1002
 Flash GetUserProgBaseAddress, 1003
 FlashInit, 1003
 FlashInitBlock, 1003
 flashLayout, 1007
 FlashOperate, 1004
 FlashSwitchBlock, 1004
 FlashVerifyChecksum, 1005
 FlashWrite, 1005
 FlashWriteBlock, 1005
 FlashWriteChecksum, 1006
FlashAddToBlock
 _template/flash.c, 226
 ARMCM0_S32K11/flash.c, 234
 ARMCM0_STM32C0/flash.c, 242
ARMCM0_STM32F0/flash.c, 251
ARMCM0_STM32G0/flash.c, 260
ARMCM0_STM32L0/flash.c, 270
ARMCM0_XMC1/flash.c, 279
ARMCM33_STM32H5/flash.c, 288
ARMCM33_STM32L5/flash.c, 299
ARMCM33_STM32U5/flash.c, 308
ARMCM3_EFM32/flash.c, 317
ARMCM3_LM3S/flash.c, 326
ARMCM3_STM32F1/flash.c, 335
ARMCM3_STM32F2/flash.c, 342
ARMCM3_STM32L1/flash.c, 350
ARMCM4_S32K14/flash.c, 359
ARMCM4_STM32F3/flash.c, 367
ARMCM4_STM32F4/flash.c, 374
ARMCM4_STM32G4/flash.c, 383
ARMCM4_STM32L4/flash.c, 392
ARMCM4_TM4C/flash.c, 400
ARMCM4_XMC4/flash.c, 409
ARMCM7_STM32F7/flash.c, 419
ARMCM7_STM32H7/flash.c, 428
flash_ecc.c, 1000
HCS12/flash.c, 437
TRICORE_TC2/flash.c, 447
TRICORE_TC3/flash.c, 457
FlashCalcPageSize
 ARMCM3_EFM32/flash.c, 318
FlashCommandSequence
 ARMCM0_S32K11/flash.c, 234
 ARMCM4_S32K14/flash.c, 359
FlashDone
 _template/flash.c, 226
 _template/flash.h, 465
 ARMCM0_S32K11/flash.c, 235
 ARMCM0_S32K11/flash.h, 469
 ARMCM0_STM32C0/flash.c, 243
 ARMCM0_STM32C0/flash.h, 473
 ARMCM0_STM32F0/flash.c, 252
 ARMCM0_STM32F0/flash.h, 477
 ARMCM0_STM32G0/flash.c, 261
 ARMCM0_STM32G0/flash.h, 481
 ARMCM0_STM32L0/flash.c, 271
 ARMCM0_STM32L0/flash.h, 485
 ARMCM0_XMC1/flash.c, 279
 ARMCM0_XMC1/flash.h, 489
 ARMCM33_STM32H5/flash.c, 288
 ARMCM33_STM32H5/flash.h, 493
 ARMCM33_STM32L5/flash.c, 299
 ARMCM33_STM32L5/flash.h, 497
 ARMCM33_STM32U5/flash.c, 309
 ARMCM33_STM32U5/flash.h, 501
 ARMCM3_EFM32/flash.c, 318
 ARMCM3_EFM32/flash.h, 505
 ARMCM3_LM3S/flash.c, 327
 ARMCM3_LM3S/flash.h, 509
 ARMCM3_STM32F1/flash.c, 335
 ARMCM3_STM32F1/flash.h, 513
 ARMCM3_STM32F2/flash.c, 343

ARMCM3_STM32F2/flash.h, 517
 ARMCM3_STM32L1/flash.c, 351
 ARMCM3_STM32L1/flash.h, 521
 ARMCM4_S32K14/flash.c, 360
 ARMCM4_S32K14/flash.h, 525
 ARMCM4_STM32F3/flash.c, 367
 ARMCM4_STM32F3/flash.h, 529
 ARMCM4_STM32F4/flash.c, 375
 ARMCM4_STM32F4/flash.h, 533
 ARMCM4_STM32G4/flash.c, 383
 ARMCM4_STM32G4/flash.h, 537
 ARMCM4_STM32L4/flash.c, 392
 ARMCM4_STM32L4/flash.h, 541
 ARMCM4_TM4C/flash.c, 400
 ARMCM4_TM4C/flash.h, 545
 ARMCM4_XMC4/flash.c, 409
 ARMCM4_XMC4/flash.h, 549
 ARMCM7_STM32F7/flash.c, 420
 ARMCM7_STM32F7/flash.h, 553
 ARMCM7_STM32H7/flash.c, 428
 ARMCM7_STM32H7/flash.h, 557
 flash_ecc.c, 1001
 HCS12/flash.c, 438
 HCS12/flash.h, 561
 TRICORE_TC2/flash.c, 448
 TRICORE_TC2/flash.h, 565
 TRICORE_TC3/flash.c, 457
 TRICORE_TC3/flash.h, 569

FlashEmptyCheckSector

ARMCM33_STM32H5/flash.c, 288
 ARMCM33_STM32U5/flash.c, 309
 ARMCM3_STM32F2/flash.c, 343
 ARMCM4_STM32F4/flash.c, 375
 ARMCM4_XMC4/flash.c, 409
 ARMCM7_STM32F7/flash.c, 420
 ARMCM7_STM32H7/flash.c, 429

FlashErase

_template/flash.c, 226
 _template/flash.h, 466
 ARMCM0_S32K11/flash.c, 235
 ARMCM0_S32K11/flash.h, 469
 ARMCM0_STM32C0/flash.c, 243
 ARMCM0_STM32C0/flash.h, 473
 ARMCM0_STM32F0/flash.c, 252
 ARMCM0_STM32F0/flash.h, 477
 ARMCM0_STM32G0/flash.c, 261
 ARMCM0_STM32G0/flash.h, 481
 ARMCM0_STM32L0/flash.c, 271
 ARMCM0_STM32L0/flash.h, 485
 ARMCM0_XMC1/flash.c, 279
 ARMCM0_XMC1/flash.h, 489
 ARMCM33_STM32H5/flash.c, 290
 ARMCM33_STM32H5/flash.h, 493
 ARMCM33_STM32L5/flash.c, 299
 ARMCM33_STM32L5/flash.h, 497
 ARMCM33_STM32U5/flash.c, 309
 ARMCM33_STM32U5/flash.h, 501
 ARMCM3_EFM32/flash.c, 318

ARMCM3_EFM32/flash.h, 505
 ARMCM3_LM3S/flash.c, 327
 ARMCM3_LM3S/flash.h, 509
 ARMCM3_STM32F1/flash.c, 336
 ARMCM3_STM32F1/flash.h, 513
 ARMCM3_STM32F2/flash.c, 343
 ARMCM3_STM32F2/flash.h, 517
 ARMCM3_STM32L1/flash.c, 351
 ARMCM3_STM32L1/flash.h, 521
 ARMCM4_S32K14/flash.c, 360
 ARMCM4_S32K14/flash.h, 525
 ARMCM4_STM32F3/flash.c, 368
 ARMCM4_STM32F3/flash.h, 529
 ARMCM4_STM32F4/flash.c, 375
 ARMCM4_STM32F4/flash.h, 533
 ARMCM4_STM32G4/flash.c, 383
 ARMCM4_STM32G4/flash.h, 537
 ARMCM4_STM32L4/flash.c, 392
 ARMCM4_STM32L4/flash.h, 541
 ARMCM4_TM4C/flash.c, 400
 ARMCM4_TM4C/flash.h, 545
 ARMCM4_XMC4/flash.c, 411
 ARMCM4_XMC4/flash.h, 549
 ARMCM7_STM32F7/flash.c, 420
 ARMCM7_STM32F7/flash.h, 553
 ARMCM7_STM32H7/flash.c, 429
 ARMCM7_STM32H7/flash.h, 557
 flash_ecc.c, 1001
 HCS12/flash.c, 438
 HCS12/flash.h, 561
 TRICORE_TC2/flash.c, 448
 TRICORE_TC2/flash.h, 565
 TRICORE_TC3/flash.c, 457
 TRICORE_TC3/flash.h, 569

FlashEraseLogicalSector

TRICORE_TC2/flash.c, 448

FlashEraseLogicalSectors

TRICORE_TC3/flash.c, 458

FlashEraseSectors

_template/flash.c, 227
 ARMCM0_S32K11/flash.c, 236
 ARMCM0_STM32C0/flash.c, 243
 ARMCM0_STM32F0/flash.c, 252
 ARMCM0_STM32G0/flash.c, 261
 ARMCM0_XMC1/flash.c, 280
 ARMCM33_STM32H5/flash.c, 290
 ARMCM33_STM32L5/flash.c, 300
 ARMCM33_STM32U5/flash.c, 310
 ARMCM3_EFM32/flash.c, 319
 ARMCM3_LM3S/flash.c, 327
 ARMCM3_STM32F2/flash.c, 344
 ARMCM4_S32K14/flash.c, 360
 ARMCM4_STM32F4/flash.c, 376
 ARMCM4_STM32G4/flash.c, 384
 ARMCM4_TM4C/flash.c, 401
 ARMCM4_XMC4/flash.c, 411
 ARMCM7_STM32F7/flash.c, 421
 ARMCM7_STM32H7/flash.c, 430

TRICORE_TC2/flash.c, 449
TRICORE_TC3/flash.c, 458
flashExecCmd
 flash_ecc.c, 1007
 HCS12/flash.c, 444
FlashExecuteCommand
 flash_ecc.c, 1001
 HCS12/flash.c, 438
FlashGetBank
 ARMCM0_STM32G0/flash.c, 262
 ARMCM33_STM32H5/flash.c, 291
 ARMCM33_STM32L5/flash.c, 300
 ARMCM33_STM32U5/flash.c, 310
 ARMCM4_STM32G4/flash.c, 384
 ARMCM4_STM32L4/flash.c, 393
FlashGetGlobalAddrByte
 flash_ecc.c, 1002
FlashGetLinearAddrByte
 HCS12/flash.c, 439
FlashGetPage
 ARMCM0_STM32C0/flash.c, 244
 ARMCM0_STM32G0/flash.c, 262
 ARMCM33_STM32H5/flash.c, 291
 ARMCM33_STM32L5/flash.c, 301
 ARMCM33_STM32U5/flash.c, 310
 ARMCM4_STM32G4/flash.c, 385
 ARMCM4_STM32L4/flash.c, 393
FlashGetPageSize
 ARMCM33_STM32L5/flash.c, 301
FlashGetPhysAddr
 flash_ecc.c, 1002
 HCS12/flash.c, 439
FlashGetPhysPage
 flash_ecc.c, 1002
 HCS12/flash.c, 440
FlashGetSector
 ARMCM0_STM32C0/flash.c, 244
 ARMCM0_STM32F0/flash.c, 253
 ARMCM0_STM32G0/flash.c, 263
 ARMCM0_XMC1/flash.c, 280
 ARMCM3_EFM32/flash.c, 319
 ARMCM3_LM3S/flash.c, 328
 ARMCM3_STM32F2/flash.c, 344
 ARMCM4_STM32F4/flash.c, 376
 ARMCM4_TM4C/flash.c, 401
 ARMCM4_XMC4/flash.c, 412
 ARMCM7_STM32F7/flash.c, 421
FlashGetSectorBaseAddr
 ARMCM0_STM32C0/flash.c, 244
 ARMCM0_STM32F0/flash.c, 253
 ARMCM0_STM32G0/flash.c, 263
 ARMCM0_XMC1/flash.c, 280
 ARMCM3_EFM32/flash.c, 319
 ARMCM3_LM3S/flash.c, 328
 ARMCM4_TM4C/flash.c, 402
 ARMCM4_XMC4/flash.c, 412
FlashGetSectorIdx
 _template/flash.c, 227
ARMCM0_S32K11/flash.c, 236
ARMCM33_STM32H5/flash.c, 291
ARMCM33_STM32L5/flash.c, 301
ARMCM33_STM32U5/flash.c, 311
ARMCM4_S32K14/flash.c, 361
ARMCM4_STM32G4/flash.c, 385
ARMCM7_STM32H7/flash.c, 430
TRICORE_TC2/flash.c, 449
TRICORE_TC3/flash.c, 460
FlashGetSectorSize
 ARMCM0_STM32C0/flash.c, 245
 ARMCM0_STM32F0/flash.c, 254
 ARMCM0_STM32G0/flash.c, 263
 ARMCM3_EFM32/flash.c, 320
 ARMCM3_LM3S/flash.c, 329
 ARMCM4_TM4C/flash.c, 402
Flash GetUserProgBaseAddress
 _template/flash.c, 227
 _template/flash.h, 466
 ARMCM0_S32K11/flash.c, 236
 ARMCM0_S32K11/flash.h, 470
 ARMCM0_STM32C0/flash.c, 245
 ARMCM0_STM32C0/flash.h, 474
 ARMCM0_STM32F0/flash.c, 254
 ARMCM0_STM32F0/flash.h, 478
 ARMCM0_STM32G0/flash.c, 264
 ARMCM0_STM32G0/flash.h, 482
 ARMCM0_STM32L0/flash.c, 272
 ARMCM0_STM32L0/flash.h, 486
 ARMCM0_XMC1/flash.c, 281
 ARMCM0_XMC1/flash.h, 490
 ARMCM33_STM32H5/flash.c, 292
 ARMCM33_STM32H5/flash.h, 494
 ARMCM33_STM32L5/flash.c, 302
 ARMCM33_STM32L5/flash.h, 498
 ARMCM33_STM32U5/flash.c, 311
 ARMCM33_STM32U5/flash.h, 502
 ARMCM3_EFM32/flash.c, 320
 ARMCM3_EFM32/flash.h, 506
 ARMCM3_LM3S/flash.c, 329
 ARMCM3_LM3S/flash.h, 510
 ARMCM3_STM32F1/flash.c, 336
 ARMCM3_STM32F1/flash.h, 514
 ARMCM3_STM32F2/flash.c, 344
 ARMCM3_STM32F2/flash.h, 518
 ARMCM3_STM32L1/flash.c, 352
 ARMCM3_STM32L1/flash.h, 522
 ARMCM4_S32K14/flash.c, 361
 ARMCM4_S32K14/flash.h, 526
 ARMCM4_STM32F3/flash.c, 368
 ARMCM4_STM32F3/flash.h, 530
 ARMCM4_STM32F4/flash.c, 376
 ARMCM4_STM32F4/flash.h, 534
 ARMCM4_STM32G4/flash.c, 385
 ARMCM4_STM32G4/flash.h, 538
 ARMCM4_STM32L4/flash.c, 393
 ARMCM4_STM32L4/flash.h, 542
 ARMCM4_TM4C/flash.c, 402

ARMCM4_TM4C/flash.h, 546
 ARMCM4_XMC4/flash.c, 412
 ARMCM4_XMC4/flash.h, 550
 ARMCM7_STM32F7/flash.c, 422
 ARMCM7_STM32F7/flash.h, 554
 ARMCM7_STM32H7/flash.c, 430
 ARMCM7_STM32H7/flash.h, 558
 flash_ecc.c, 1003
 HCS12/flash.c, 440
 HCS12/flash.h, 562
 TRICORE_TC2/flash.c, 450
 TRICORE_TC2/flash.h, 566
 TRICORE_TC3/flash.c, 460
 TRICORE_TC3/flash.h, 570

FlashInit
 _template/flash.c, 228
 _template/flash.h, 467
 ARMCM0_S32K11/flash.c, 237
 ARMCM0_S32K11/flash.h, 470
 ARMCM0_STM32C0/flash.c, 245
 ARMCM0_STM32C0/flash.h, 474
 ARMCM0_STM32F0/flash.c, 254
 ARMCM0_STM32F0/flash.h, 478
 ARMCM0_STM32G0/flash.c, 264
 ARMCM0_STM32G0/flash.h, 482
 ARMCM0_STM32L0/flash.c, 272
 ARMCM0_STM32L0/flash.h, 486
 ARMCM0_XMC1/flash.c, 281
 ARMCM0_XMC1/flash.h, 490
 ARMCM33_STM32H5/flash.c, 292
 ARMCM33_STM32H5/flash.h, 494
 ARMCM33_STM32L5/flash.c, 302
 ARMCM33_STM32L5/flash.h, 498
 ARMCM33_STM32U5/flash.c, 311
 ARMCM33_STM32U5/flash.h, 502
 ARMCM3_EFM32/flash.c, 320
 ARMCM3_EFM32/flash.h, 506
 ARMCM3_LM3S/flash.c, 329
 ARMCM3_LM3S/flash.h, 510
 ARMCM3_STM32F1/flash.c, 336
 ARMCM3_STM32F1/flash.h, 514
 ARMCM3_STM32F2/flash.c, 345
 ARMCM3_STM32F2/flash.h, 518
 ARMCM3_STM32L1/flash.c, 352
 ARMCM3_STM32L1/flash.h, 522
 ARMCM4_S32K14/flash.c, 361
 ARMCM4_S32K14/flash.h, 526
 ARMCM4_STM32F3/flash.c, 368
 ARMCM4_STM32F3/flash.h, 530
 ARMCM4_STM32F4/flash.c, 377
 ARMCM4_STM32F4/flash.h, 534
 ARMCM4_STM32G4/flash.c, 386
 ARMCM4_STM32G4/flash.h, 538
 ARMCM4_STM32L4/flash.c, 394
 ARMCM4_STM32L4/flash.h, 542
 ARMCM4_TM4C/flash.c, 403
 ARMCM4_TM4C/flash.h, 546
 ARMCM4_XMC4/flash.c, 413

ARMCM4_XMC4/flash.h, 550
 ARMCM7_STM32F7/flash.c, 422
 ARMCM7_STM32F7/flash.h, 554
 ARMCM7_STM32H7/flash.c, 431
 ARMCM7_STM32H7/flash.h, 558
 flash_ecc.c, 1003
 HCS12/flash.c, 440
 HCS12/flash.h, 562
 TRICORE_TC2/flash.c, 450
 TRICORE_TC2/flash.h, 566
 TRICORE_TC3/flash.c, 460
 TRICORE_TC3/flash.h, 570

FlashInitBlock
 _template/flash.c, 228
 ARMCM0_S32K11/flash.c, 237
 ARMCM0_STM32C0/flash.c, 246
 ARMCM0_STM32F0/flash.c, 254
 ARMCM0_STM32G0/flash.c, 264
 ARMCM0_STM32L0/flash.c, 272
 ARMCM0_XMC1/flash.c, 281
 ARMCM33_STM32H5/flash.c, 292
 ARMCM33_STM32L5/flash.c, 302
 ARMCM33_STM32U5/flash.c, 312
 ARMCM3_EFM32/flash.c, 321
 ARMCM3_LM3S/flash.c, 329
 ARMCM3_STM32F1/flash.c, 337
 ARMCM3_STM32F2/flash.c, 345
 ARMCM3_STM32L1/flash.c, 352
 ARMCM4_S32K14/flash.c, 362
 ARMCM4_STM32F3/flash.c, 369
 ARMCM4_STM32F4/flash.c, 377
 ARMCM4_STM32G4/flash.c, 386
 ARMCM4_STM32L4/flash.c, 394
 ARMCM4_TM4C/flash.c, 403
 ARMCM4_XMC4/flash.c, 413
 ARMCM7_STM32F7/flash.c, 422
 ARMCM7_STM32H7/flash.c, 431
 flash_ecc.c, 1003
 HCS12/flash.c, 440
 TRICORE_TC2/flash.c, 450
 TRICORE_TC3/flash.c, 461

FlashIsDualBankMode
 ARMCM33_STM32L5/flash.c, 303

FlashIsSingleBankMode
 ARMCM7_STM32F7/flash.c, 423

flashLayout
 _template/flash.c, 231
 ARMCM0_STM32C0/flash.c, 249
 ARMCM0_STM32F0/flash.c, 257
 ARMCM0_STM32G0/flash.c, 268
 ARMCM0_STM32L0/flash.c, 276
 ARMCM0_XMC1/flash.c, 285
 ARMCM33_STM32H5/flash.c, 296
 ARMCM33_STM32L5/flash.c, 305
 ARMCM33_STM32U5/flash.c, 315
 ARMCM3_EFM32/flash.c, 324
 ARMCM3_LM3S/flash.c, 332
 ARMCM3_STM32F1/flash.c, 340

ARMCM3_STM32F2/flash.c, 348
ARMCM3_STM32L1/flash.c, 356
ARMCM4_STM32F3/flash.c, 372
ARMCM4_STM32F4/flash.c, 380
ARMCM4_STM32G4/flash.c, 389
ARMCM4_STM32L4/flash.c, 397
ARMCM4_TM4C/flash.c, 406
ARMCM4_XMC4/flash.c, 416
ARMCM7_STM32F7/flash.c, 425
ARMCM7_STM32H7/flash.c, 434
flash_ecc.c, 1007
HCS12/flash.c, 444
TRICORE_TC2/flash.c, 454
TRICORE_TC3/flash.c, 464

FlashOperate
 flash_ecc.c, 1004
 HCS12/flash.c, 441

FlashSwitchBlock
 _template/flash.c, 228
 ARMCM0_S32K11/flash.c, 237
 ARMCM0_STM32C0/flash.c, 246
 ARMCM0_STM32F0/flash.c, 255
 ARMCM0_STM32G0/flash.c, 265
 ARMCM0_STM32L0/flash.c, 273
 ARMCM0_XMC1/flash.c, 282
 ARMCM33_STM32H5/flash.c, 293
 ARMCM33_STM32L5/flash.c, 303
 ARMCM33_STM32U5/flash.c, 313
 ARMCM33_STM32U5/flash.h, 503
 ARMCM3_EFM32/flash.c, 322
 ARMCM3_EFM32/flash.h, 507
 ARMCM3_LM3S/flash.c, 330
 ARMCM3_LM3S/flash.h, 511
 ARMCM3_STM32F1/flash.c, 338
 ARMCM3_STM32F1/flash.h, 515
 ARMCM3_STM32F2/flash.c, 346
 ARMCM3_STM32F2/flash.h, 519
 ARMCM3_STM32L1/flash.c, 353
 ARMCM3_STM32L1/flash.h, 523
 ARMCM4_S32K14/flash.c, 363
 ARMCM4_S32K14/flash.h, 527
 ARMCM4_STM32F3/flash.c, 370
 ARMCM4_STM32F3/flash.h, 531
 ARMCM4_STM32F4/flash.c, 378
 ARMCM4_STM32F4/flash.h, 535
 ARMCM4_STM32G4/flash.c, 387
 ARMCM4_STM32G4/flash.h, 539
 ARMCM4_STM32L4/flash.c, 395
 ARMCM4_STM32L4/flash.h, 543
 ARMCM4_TM4C/flash.c, 404
 ARMCM4_TM4C/flash.h, 547
 ARMCM4_XMC4/flash.c, 414
 ARMCM4_XMC4/flash.h, 551
 ARMCM7_STM32F7/flash.c, 423
 ARMCM7_STM32F7/flash.h, 555
 ARMCM7_STM32H7/flash.c, 432
 ARMCM7_STM32H7/flash.h, 559
 flash_ecc.c, 1005
 HCS12/flash.c, 442
 HCS12/flash.h, 563
 TRICORE_TC2/flash.c, 452
 TRICORE_TC2/flash.h, 567
 TRICORE_TC3/flash.c, 462
 TRICORE_TC3/flash.h, 571

FlashTranslateToNonCachedAddress
 ARMCM4_XMC4/flash.c, 414
 TRICORE_TC2/flash.c, 451

FlashVerifyBankMode
 ARMCM4_STM32G4/flash.c, 387

FlashVerifyChecksum
 _template/flash.c, 229
 _template/flash.h, 467
 ARMCM0_S32K11/flash.c, 238
 ARMCM0_S32K11/flash.h, 471
 ARMCM0_STM32C0/flash.c, 247
 ARMCM0_STM32C0/flash.h, 475

FlashWrite
 _template/flash.c, 229
 _template/flash.h, 467
 ARMCM0_S32K11/flash.c, 238
 ARMCM0_S32K11/flash.h, 471
 ARMCM0_STM32C0/flash.c, 247
 ARMCM0_STM32C0/flash.h, 475
 ARMCM0_STM32F0/flash.c, 255
 ARMCM0_STM32F0/flash.h, 479

ARMCM0_STM32G0/flash.c, 265
 ARMCM0_STM32G0/flash.h, 483
 ARMCM0_STM32L0/flash.c, 273
 ARMCM0_STM32L0/flash.h, 487
 ARMCM0_XMC1/flash.c, 282
 ARMCM0_XMC1/flash.h, 491
 ARMCM33_STM32H5/flash.c, 293
 ARMCM33_STM32H5/flash.h, 495
 ARMCM33_STM32L5/flash.c, 304
 ARMCM33_STM32L5/flash.h, 499
 ARMCM33_STM32U5/flash.c, 313
 ARMCM33_STM32U5/flash.h, 503
 ARMCM3_EFM32/flash.c, 322
 ARMCM3_EFM32/flash.h, 507
 ARMCM3_LM3S/flash.c, 330
 ARMCM3_LM3S/flash.h, 511
 ARMCM3_STM32F1/flash.c, 338
 ARMCM3_STM32F1/flash.h, 515
 ARMCM3_STM32F2/flash.c, 346
 ARMCM3_STM32F2/flash.h, 519
 ARMCM3_STM32L1/flash.c, 353
 ARMCM3_STM32L1/flash.h, 523
 ARMCM4_S32K14/flash.c, 363
 ARMCM4_S32K14/flash.h, 527
 ARMCM4_STM32F3/flash.c, 370
 ARMCM4_STM32F3/flash.h, 531
 ARMCM4_STM32F4/flash.c, 378
 ARMCM4_STM32F4/flash.h, 535
 ARMCM4_STM32G4/flash.c, 387
 ARMCM4_STM32G4/flash.h, 539
 ARMCM4_STM32L4/flash.c, 395
 ARMCM4_STM32L4/flash.h, 543
 ARMCM4_TM4C/flash.c, 404
 ARMCM4_TM4C/flash.h, 547
 ARMCM4_XMC4/flash.c, 414
 ARMCM4_XMC4/flash.h, 551
 ARMCM7_STM32F7/flash.c, 424
 ARMCM7_STM32F7/flash.h, 555
 ARMCM7_STM32H7/flash.c, 432
 ARMCM7_STM32H7/flash.h, 559
 flash_ecc.c, 1005
 HCS12/flash.c, 442
 HCS12/flash.h, 563
 TRICORE_TC2/flash.c, 452
 TRICORE_TC2/flash.h, 567
 TRICORE_TC3/flash.c, 462
 TRICORE_TC3/flash.h, 571

FlashWriteBlock
 _template/flash.c, 230
 ARMCM0_S32K11/flash.c, 238
 ARMCM0_STM32C0/flash.c, 247
 ARMCM0_STM32F0/flash.c, 256
 ARMCM0_STM32G0/flash.c, 267
 ARMCM0_STM32L0/flash.c, 275
 ARMCM0_XMC1/flash.c, 284
 ARMCM33_STM32H5/flash.c, 295
 ARMCM33_STM32H5/flash.h, 496
 ARMCM33_STM32L5/flash.c, 305
 ARMCM33_STM32L5/flash.h, 500
 ARMCM33_STM32U5/flash.c, 314
 ARMCM33_STM32U5/flash.h, 504
 ARMCM3_EFM32/flash.c, 323
 ARMCM3_EFM32/flash.h, 508
 ARMCM3_LM3S/flash.c, 331
 ARMCM3_LM3S/flash.h, 512
 ARMCM3_STM32F1/flash.c, 339
 ARMCM3_STM32F1/flash.h, 516
 ARMCM3_STM32F2/flash.c, 347
 ARMCM3_STM32F2/flash.h, 520
 ARMCM3_STM32L1/flash.c, 355
 ARMCM3_STM32L1/flash.h, 524
 ARMCM4_S32K14/flash.c, 364
 ARMCM4_S32K14/flash.h, 528
 ARMCM4_STM32F3/flash.c, 371
 ARMCM4_STM32F3/flash.h, 532
 ARMCM4_STM32F4/flash.c, 379
 ARMCM4_STM32F4/flash.h, 536
 ARMCM4_STM32G4/flash.c, 388
 ARMCM4_STM32G4/flash.h, 540
 ARMCM4_STM32L4/flash.c, 396

ARMCM4_STM32L4/flash.h, 544
ARMCM4_TM4C/flash.c, 405
ARMCM4_TM4C/flash.h, 548
ARMCM4_XMC4/flash.c, 415
ARMCM4_XMC4/flash.h, 552
ARMCM7_STM32F7/flash.c, 425
ARMCM7_STM32F7/flash.h, 556
ARMCM7_STM32H7/flash.c, 433
ARMCM7_STM32H7/flash.h, 560
flash_ecc.c, 1006
HCS12/flash.c, 443
HCS12/flash.h, 564
TRICORE_TC2/flash.c, 453
TRICORE_TC2/flash.h, 568
TRICORE_TC3/flash.c, 463
TRICORE_TC3/flash.h, 572
FlashWritePage
 TRICORE_TC2/flash.c, 453
 TRICORE_TC3/flash.c, 463
fopt
 tFlashRegs, 70
fprot
 tFlashRegs, 70
frsv0
 tFlashRegs, 71
frsv1
 tFlashRegs, 71
frsv2
 tFlashRegs, 71
frsv3
 tFlashRegs, 71
frsv4
 tFlashRegs, 71
frsv5
 tFlashRegs, 71
frsv6
 tFlashRegs, 72
frsv7
 tFlashRegs, 72
fs
 tFatFsObjects, 62
fsec
 tFlashRegs, 72
fstat
 tFlashRegs, 72
ftstmod
 tFlashRegs, 72
HAL_GetTick
 ARMCM0_STM32C0/timer.c, 770
 ARMCM0_STM32F0/timer.c, 773
 ARMCM0_STM32G0/timer.c, 775
 ARMCM0_STM32L0/timer.c, 778
 ARMCM33_STM32H5/timer.c, 783
 ARMCM33_STM32L5/timer.c, 786
 ARMCM33_STM32U5/timer.c, 788
 ARMCM3_STM32F1/timer.c, 795
 ARMCM3_STM32F2/timer.c, 798
 ARMCM3_STM32L1/timer.c, 800
 ARMCM4_STM32F3/timer.c, 805
 ARMCM4_STM32F4/timer.c, 808
 ARMCM4_STM32G4/timer.c, 810
 ARMCM4_STM32L4/timer.c, 813
 ARMCM7_STM32F7/timer.c, 820
 ARMCM7_STM32H7/timer.c, 823
handle
 tFifoCtrl, 64
 tFifoPipe, 65
HCS12/can.c
 CanGetSpeedConfig, 150
 CanInit, 150
 CanReceivePacket, 150
 canTiming, 151
 CanTransmitPacket, 151
HCS12/CodeWarrior/cpu_comp.c
 CpuIrqDisable, 653
 CpuIrqEnable, 653
HCS12/cpu.c
 CPU_USER_PROGRAM_STARTADDR_PTR, 216
 CpuInit, 217
 CpuMemcpy, 217
 CpuMemSet, 217
 CpuStartUserProgram, 218
HCS12/flash.c
 blockInfo, 443
 bootBlockInfo, 443
 FlashAddToBlock, 437
 FlashDone, 438
 FlashErase, 438
 flashExecCmd, 444
 FlashExecuteCommand, 438
 FlashGetLinearAddrByte, 439
 FlashGetPhysAddr, 439
 FlashGetPhysPage, 440
 Flash GetUserProgBaseAddress, 440
 FlashInit, 440
 FlashInitBlock, 440
 flashLayout, 444
 FlashOperate, 441
 FlashSwitchBlock, 441
 FlashVerifyChecksum, 442
 FlashWrite, 442
 FlashWriteBlock, 442
 FlashWriteChecksum, 443
HCS12/flash.h
 FlashDone, 561
 FlashErase, 561
 Flash GetUserProgBaseAddress, 562
 FlashInit, 562
 FlashVerifyChecksum, 563
 FlashWrite, 563
 FlashWriteChecksum, 564
HCS12/nvm.c
 NvmDone, 742
 NvmErase, 743
 Nvm GetUserProgBaseAddress, 743
 NvmlInit, 743

NvmVerifyChecksum, 744
 NvmWrite, 744
 HCS12/timer.c
 TimerGet, 825
 TimerInit, 826
 TimerReset, 826
 TimerUpdate, 826
 HCS12/types.h
 blt_addr, 883
 blt_bool, 883
 blt_char, 883
 blt_int16s, 883
 blt_int16u, 883
 blt_int32s, 884
 blt_int32u, 884
 blt_int8s, 884
 blt_int8u, 884
 idr
 tCanRxMsgSlot, 60
 tCanTxMsgSlot, 61
 length
 tFifoCtrl, 64
 line
 tSrecLineParseObject, 75
 LINE_TYPE_S1
 file.h, 994
 LINE_TYPE_S2
 file.h, 994
 LINE_TYPE_S3
 file.h, 994
 LINE_TYPE_UNSUPPORTED
 file.h, 994
 mb.c, 1008
 mb.h, 1008
 mbrtu.c, 656–669
 Modbus RTU driver of a port, 23
 mta
 tXcpInfo, 80
 net.c, 1009
 NetApp, 1010
 NetInit, 1010
 NetReceivePacket, 1010
 NetServerTask, 1011
 NetTransmitPacket, 1011
 net.h, 1012
 NetApp, 1012
 NetInit, 1013
 NetReceivePacket, 1013
 NetTransmitPacket, 1013
 NetApp
 net.c, 1010
 net.h, 1012
 NetInit
 net.c, 1010
 net.h, 1013
 NetReceivePacket
 net.c, 1010
 net.h, 1013
 NetServerTask
 net.c, 1011
 NetTransmitPacket
 net.c, 1011
 net.h, 1013
 Non-volatile memory driver of a port, 23
 nvm.c, 669, 673, 676, 678, 681, 684, 687, 690, 693, 696, 699, 702, 705, 708, 711, 714, 717, 720, 723, 726, 729, 732, 735, 738, 741, 744, 747
 nvm.h, 1014
 NvmDone, 1015
 NvmErase, 1015
 Nvm GetUserProgBaseAddress, 1015
 NvmlInit, 1016
 NvmVerifyChecksum, 1016
 NvmWrite, 1016
 NvmDone
 \$template/nvm.c, 670
 ARMCM0_S32K11/nvm.c, 674
 ARMCM0_STM32C0/nvm.c, 676
 ARMCM0_STM32F0/nvm.c, 679
 ARMCM0_STM32G0/nvm.c, 682
 ARMCM0_STM32L0/nvm.c, 685
 ARMCM0_XMC1/nvm.c, 688
 ARMCM33_STM32H5/nvm.c, 691
 ARMCM33_STM32L5/nvm.c, 694
 ARMCM33_STM32U5/nvm.c, 697
 ARMCM3_EFM32/nvm.c, 700
 ARMCM3_LM3S/nvm.c, 703
 ARMCM3_STM32F1/nvm.c, 706
 ARMCM3_STM32F2/nvm.c, 709
 ARMCM3_STM32L1/nvm.c, 712
 ARMCM4_S32K14/nvm.c, 715
 ARMCM4_STM32F3/nvm.c, 718
 ARMCM4_STM32F4/nvm.c, 721
 ARMCM4_STM32G4/nvm.c, 724
 ARMCM4_STM32L4/nvm.c, 727
 ARMCM4_TM4C/nvm.c, 730
 ARMCM4_XMC4/nvm.c, 733
 ARMCM7_STM32F7/nvm.c, 736
 ARMCM7_STM32H7/nvm.c, 739
 HCS12/nvm.c, 742
 nvm.h, 1015
 TRICORE_TC2/nvm.c, 745
 TRICORE_TC3/nvm.c, 748
 NvmErase
 \$template/nvm.c, 670
 ARMCM0_S32K11/nvm.c, 674
 ARMCM0_STM32C0/nvm.c, 677
 ARMCM0_STM32F0/nvm.c, 680
 ARMCM0_STM32G0/nvm.c, 683
 ARMCM0_STM32L0/nvm.c, 686
 ARMCM0_XMC1/nvm.c, 689
 ARMCM33_STM32H5/nvm.c, 692
 ARMCM33_STM32L5/nvm.c, 695

ARMCM33_STM32U5/nvm.c, 698
ARMCM3_EFM32/nvm.c, 701
ARMCM3_LM3S/nvm.c, 704
ARMCM3_STM32F1/nvm.c, 707
ARMCM3_STM32F2/nvm.c, 710
ARMCM3_STM32L1/nvm.c, 713
ARMCM4_S32K14/nvm.c, 716
ARMCM4_STM32F3/nvm.c, 719
ARMCM4_STM32F4/nvm.c, 722
ARMCM4_STM32G4/nvm.c, 725
ARMCM4_STM32L4/nvm.c, 728
ARMCM4_TM4C/nvm.c, 731
ARMCM4_XMC4/nvm.c, 734
ARMCM7_STM32F7/nvm.c, 737
ARMCM7_STM32H7/nvm.c, 740
HCS12/nvm.c, 743
nvm.h, 1015
TRICORE_TC2/nvm.c, 746
TRICORE_TC3/nvm.c, 749

Nvm GetUserProgBaseAddress
 _template/nvm.c, 671
 ARMCM0_S32K11/nvm.c, 674
 ARMCM0_STM32C0/nvm.c, 677
 ARMCM0_STM32F0/nvm.c, 680
 ARMCM0_STM32G0/nvm.c, 683
 ARMCM0_STM32L0/nvm.c, 686
 ARMCM0_XMC1/nvm.c, 689
 ARMCM33_STM32H5/nvm.c, 692
 ARMCM33_STM32L5/nvm.c, 695
 ARMCM33_STM32U5/nvm.c, 698
 ARMCM3_EFM32/nvm.c, 701
 ARMCM3_LM3S/nvm.c, 704
 ARMCM3_STM32F1/nvm.c, 707
 ARMCM3_STM32F2/nvm.c, 710
 ARMCM3_STM32L1/nvm.c, 713
 ARMCM4_S32K14/nvm.c, 716
 ARMCM4_STM32F3/nvm.c, 719
 ARMCM4_STM32F4/nvm.c, 722
 ARMCM4_STM32G4/nvm.c, 725
 ARMCM4_STM32L4/nvm.c, 728
 ARMCM4_TM4C/nvm.c, 731
 ARMCM4_XMC4/nvm.c, 734
 ARMCM7_STM32F7/nvm.c, 737
 ARMCM7_STM32H7/nvm.c, 740
 HCS12/nvm.c, 743
 nvm.h, 1015
 TRICORE_TC2/nvm.c, 746
 TRICORE_TC3/nvm.c, 749

NvmInit
 _template/nvm.c, 671
 ARMCM0_S32K11/nvm.c, 675
 ARMCM0_STM32C0/nvm.c, 677
 ARMCM0_STM32F0/nvm.c, 680
 ARMCM0_STM32G0/nvm.c, 683
 ARMCM0_STM32L0/nvm.c, 686
 ARMCM0_XMC1/nvm.c, 689
 ARMCM33_STM32H5/nvm.c, 692
 ARMCM33_STM32L5/nvm.c, 695

NvmVerifyChecksum
 _template/nvm.c, 671
 ARMCM0_S32K11/nvm.c, 675
 ARMCM0_STM32C0/nvm.c, 678
 ARMCM0_STM32F0/nvm.c, 681
 ARMCM0_STM32G0/nvm.c, 684
 ARMCM0_STM32L0/nvm.c, 687
 ARMCM0_XMC1/nvm.c, 690
 ARMCM33_STM32H5/nvm.c, 693
 ARMCM33_STM32L5/nvm.c, 696
 ARMCM33_STM32U5/nvm.c, 699
 ARMCM3_EFM32/nvm.c, 702
 ARMCM3_LM3S/nvm.c, 705
 ARMCM3_STM32F1/nvm.c, 708
 ARMCM3_STM32F2/nvm.c, 711
 ARMCM3_STM32L1/nvm.c, 714
 ARMCM4_S32K14/nvm.c, 717
 ARMCM4_STM32F3/nvm.c, 720
 ARMCM4_STM32F4/nvm.c, 723
 ARMCM4_STM32G4/nvm.c, 726
 ARMCM4_STM32L4/nvm.c, 729
 ARMCM4_TM4C/nvm.c, 732
 ARMCM4_XMC4/nvm.c, 735
 ARMCM7_STM32F7/nvm.c, 738
 ARMCM7_STM32H7/nvm.c, 741
 HCS12/nvm.c, 744
 nvm.h, 1016
 TRICORE_TC2/nvm.c, 747
 TRICORE_TC3/nvm.c, 750

NvmWrite
 _template/nvm.c, 671
 ARMCM0_S32K11/nvm.c, 675
 ARMCM0_STM32C0/nvm.c, 678
 ARMCM0_STM32F0/nvm.c, 681
 ARMCM0_STM32G0/nvm.c, 684
 ARMCM0_STM32L0/nvm.c, 687
 ARMCM0_XMC1/nvm.c, 690
 ARMCM33_STM32H5/nvm.c, 693
 ARMCM33_STM32L5/nvm.c, 696

ARMCM33_STM32U5/nvm.c, 699
 ARMCM3_EFM32/nvm.c, 702
 ARMCM3_LM3S/nvm.c, 705
 ARMCM3_STM32F1/nvm.c, 708
 ARMCM3_STM32F2/nvm.c, 711
 ARMCM3_STM32L1/nvm.c, 714
 ARMCM4_S32K14/nvm.c, 717
 ARMCM4_STM32F3/nvm.c, 720
 ARMCM4_STM32F4/nvm.c, 723
 ARMCM4_STM32G4/nvm.c, 726
 ARMCM4_STM32L4/nvm.c, 729
 ARMCM4_TM4C/nvm.c, 732
 ARMCM4_XMC4/nvm.c, 735
 ARMCM7_STM32F7/nvm.c, 738
 ARCM7_STM32H7/nvm.c, 741
 HCS12/nvm.c, 744
 nvm.h, 1016
 TRICORE_TC2/nvm.c, 747
 TRICORE_TC3/nvm.c, 750

oc7d
 tTimerRegs, 76

oc7m
 tTimerRegs, 76

phaseSeg1
 tCanBusTiming, 51

phaseSeg2
 tCanBusTiming, 51

plausibility.h, 1017

prescaler
 tFlashPrescalerSysclockMapping, 68

propSeg
 tCanBusTiming, 52

protection
 tXcpInfo, 80

ram_func.h, 1020, 1021

readptr
 tFifoCtrl, 64

RS232 UART driver of a port, 23

rs232.c, 750–764

rs232.h, 1017

rxSlot
 tCanRegs, 59

s_n_k_resource
 tXcpInfo, 80

sector_num
 tFlashSector, 73

sector_size
 tFlashSector, 73

sector_start
 tFlashSector, 74

start_address
 tFileEraseInfo, 66

startptr
 tFifoCtrl, 64

sysclock_max

tFlashPrescalerSysclockMapping, 68
 sysclock_min
 tFlashPrescalerSysclockMapping, 68

Target ARMCM0 S32K11, 26
 Target ARMCM0 STM32C0, 26
 Target ARMCM0 STM32F0, 27
 Target ARMCM0 STM32G0, 28
 Target ARMCM0 STM32L0, 29
 Target ARMCM0 XMC1, 29
 Target ARMCM3 EFM32, 33
 Target ARMCM3 LM3S, 33
 Target ARMCM3 STM32F1, 34
 Target ARMCM3 STM32F2, 35
 Target ARMCM3 STM32L1, 36
 Target ARMCM33 STM32H5, 30
 Target ARMCM33 STM32L5, 31
 Target ARMCM33 STM32U5, 32
 Target ARMCM4 S32K14, 37
 Target ARMCM4 STM32F3, 37
 Target ARMCM4 STM32F4, 38
 Target ARMCM4 STM32G4, 39
 Target ARMCM4 STM32L4, 40
 Target ARMCM4 TM4C, 41
 Target ARMCM4 XMC4, 42
 Target ARMCM7 STM32F7, 43
 Target ARMCM7 STM32H7, 43
 Target HCS12, 46
 Target Port Template, 24
 Target TRICORE TC2, 48
 Target TRICORE TC3, 49

tbpr
 tCanTxMsgSlot, 61

tc
 tTimerRegs, 77
 tCanBusTiming, 51
 phaseSeg1, 51
 phaseSeg2, 51
 propSeg, 52
 timeQuanta, 52
 tseg1, 52
 tseg2, 52

tCanRegs, 53
 cbtr0, 54
 cbtr1, 54
 cctl0, 55
 cctl1, 55
 cidac, 55
 cedar0, 55
 cedar1, 55
 cedar2, 55
 cedar3, 55
 cedar4, 56
 cedar5, 56
 cedar6, 56
 cedar7, 56
 cidmr0, 56
 cidmr1, 56
 cidmr2, 56

cidmr3, 57
cidmr4, 57
cidmr5, 57
cidmr6, 57
cidmr7, 57
crlg, 57
crier, 57
crxerr, 58
ctaak, 58
ctarq, 58
ctbsel, 58
ctflg, 58
ctier, 58
ctxerr, 58
dummy1, 59
rxSlot, 59
txSlot, 59
tCanRxMsgSlot, 59
 dlr, 60
 dsr, 60
 dummy, 60
 idr, 60
 tstamp, 60
tCanTxMsgSlot, 60
 dlr, 61
 dsr, 61
 idr, 61
 tbpr, 61
 tstamp, 61
tcnt
 tTimerRegs, 77
tComInterfaceld
 com.h, 978
tctl1
 tTimerRegs, 77
tctl2
 tTimerRegs, 77
tctl3
 tTimerRegs, 77
tctl4
 tTimerRegs, 77
tFatFsObjects, 62
 file, 62
 fs, 62
tFifoCtrl, 63
 endptr, 63
 entries, 63
 fifoctrlptr, 63
 handle, 64
 length, 64
 readptr, 64
 startptr, 64
 writeptr, 64
tFifoPipe, 65
 data, 65
 handle, 65
tFileEraseInfo, 66
 start_address, 66
 total_size, 66
tFirmwareUpdateState
 file.c, 988
tFlashBlockInfo, 67
tFlashPrescalerSysclockMapping, 68
 prescaler, 68
 sysclock_max, 68
 sysclock_min, 68
tFlashRegs, 69
 dfprot, 69
 fccob, 69
 fccobix, 69
 fclkdiv, 70
 fcmd, 70
 fcnfg, 70
 fercnfg, 70
 ferstat, 70
 fopt, 70
 fprot, 70
 frsv0, 71
 frsv1, 71
 frsv2, 71
 frsv3, 71
 frsv4, 71
 frsv5, 71
 frsv6, 72
 frsv7, 72
 fsec, 72
 fstat, 72
 ftstmod, 72
tFlashSector, 73
 bank_num, 73
 sector_num, 73
 sector_size, 73
 sector_start, 74
tf1g1
 tTimerRegs, 77
tf1g2
 tTimerRegs, 78
tie
 tTimerRegs, 78
timeQuanta
 tCanBusTiming, 52
Timer driver of a port, 24
timer.c, 764, 767, 769, 772, 774, 777, 780, 782, 785, 787, 790, 792, 794, 797, 799, 802, 804, 807, 809, 812, 815, 817, 819, 822, 824, 827, 829
timer.h, 1017
 TimerGet, 1018
 TimerInit, 1018
 TimerReset, 1019
 TimerUpdate, 1020
TimerGet
 _template/timer.c, 765
 ARMCM0_S32K11/timer.c, 767
 ARMCM0_STM32C0/timer.c, 770
 ARMCM0_STM32F0/timer.c, 773
 ARMCM0_STM32G0/timer.c, 775

ARMCM0_STM32L0/timer.c, 778
 ARMCM0_XMC1/timer.c, 780
 ARMCM33_STM32H5/timer.c, 783
 ARMCM33_STM32L5/timer.c, 786
 ARMCM33_STM32U5/timer.c, 788
 ARMCM3_EFM32/timer.c, 791
 ARMCM3_LM3S/timer.c, 793
 ARMCM3_STM32F1/timer.c, 795
 ARMCM3_STM32F2/timer.c, 798
 ARMCM3_STM32L1/timer.c, 800
 ARMCM4_S32K14/timer.c, 803
 ARMCM4_STM32F3/timer.c, 805
 ARMCM4_STM32F4/timer.c, 808
 ARMCM4_STM32G4/timer.c, 810
 ARMCM4_STM32L4/timer.c, 813
 ARMCM4_TM4C/timer.c, 815
 ARMCM4_XMC4/timer.c, 817
 ARMCM7_STM32F7/timer.c, 820
 ARMCM7_STM32H7/timer.c, 823
 HCS12/timer.c, 825
 timer.h, 1018
 TRICORE_TC2/timer.c, 827
 TRICORE_TC3/timer.c, 829
TimerInit
 _template/timer.c, 765
 ARMCM0_S32K11/timer.c, 768
 ARMCM0_STM32C0/timer.c, 770
 ARMCM0_STM32F0/timer.c, 773
 ARMCM0_STM32G0/timer.c, 776
 ARMCM0_STM32L0/timer.c, 778
 ARMCM0_XMC1/timer.c, 781
 ARMCM33_STM32H5/timer.c, 783
 ARMCM33_STM32L5/timer.c, 786
 ARMCM33_STM32U5/timer.c, 789
 ARMCM3_EFM32/timer.c, 791
 ARMCM3_LM3S/timer.c, 793
 ARMCM3_STM32F1/timer.c, 795
 ARMCM3_STM32F2/timer.c, 798
 ARMCM3_STM32L1/timer.c, 801
 ARMCM4_S32K14/timer.c, 803
 ARMCM4_STM32F3/timer.c, 805
 ARMCM4_STM32F4/timer.c, 808
 ARMCM4_STM32G4/timer.c, 811
 ARMCM4_STM32L4/timer.c, 813
 ARMCM4_TM4C/timer.c, 816
 ARMCM4_XMC4/timer.c, 818
 ARMCM7_STM32F7/timer.c, 820
 ARMCM7_STM32H7/timer.c, 823
 HCS12/timer.c, 826
 timer.h, 1018
 TRICORE_TC2/timer.c, 828
 TRICORE_TC3/timer.c, 830
TimerReset
 _template/timer.c, 766
 ARMCM0_S32K11/timer.c, 768
 ARMCM0_STM32C0/timer.c, 771
 ARMCM0_STM32F0/timer.c, 773
 ARMCM0_STM32G0/timer.c, 776
ARMCM0_STM32L0/timer.c, 779
ARMCM0_XMC1/timer.c, 781
ARMCM33_STM32H5/timer.c, 784
ARMCM33_STM32L5/timer.c, 786
ARMCM33_STM32U5/timer.c, 789
ARMCM3_EFM32/timer.c, 791
ARMCM3_LM3S/timer.c, 793
ARMCM3_STM32F1/timer.c, 796
ARMCM3_STM32F2/timer.c, 798
ARMCM3_STM32L1/timer.c, 801
ARMCM4_S32K14/timer.c, 803
ARMCM4_STM32F3/timer.c, 806
ARMCM4_STM32F4/timer.c, 808
ARMCM4_STM32G4/timer.c, 811
ARMCM4_STM32L4/timer.c, 814
ARMCM4_TM4C/timer.c, 816
ARMCM4_XMC4/timer.c, 818
ARMCM7_STM32F7/timer.c, 821
ARMCM7_STM32H7/timer.c, 823
HCS12/timer.c, 826
timer.h, 1019
TRICORE_TC2/timer.c, 828
TRICORE_TC3/timer.c, 830
TimerUpdate
 _template/timer.c, 766
 ARMCM0_S32K11/timer.c, 768
 ARMCM0_STM32C0/timer.c, 771
 ARMCM0_STM32F0/timer.c, 774
 ARMCM0_STM32G0/timer.c, 776
 ARMCM0_STM32L0/timer.c, 779
 ARMCM0_XMC1/timer.c, 781
 ARMCM33_STM32H5/timer.c, 784
 ARMCM33_STM32L5/timer.c, 787
 ARMCM33_STM32U5/timer.c, 789
 ARMCM3_EFM32/timer.c, 791
 ARMCM3_LM3S/timer.c, 793
 ARMCM3_STM32F1/timer.c, 796
 ARMCM3_STM32F2/timer.c, 799
 ARMCM3_STM32L1/timer.c, 801
 ARMCM4_S32K14/timer.c, 803
 ARMCM4_STM32F3/timer.c, 806
 ARMCM4_STM32F4/timer.c, 809
 ARMCM4_STM32G4/timer.c, 811
 ARMCM4_STM32L4/timer.c, 814
 ARMCM4_TM4C/timer.c, 816
 ARMCM4_XMC4/timer.c, 818
 ARMCM7_STM32F7/timer.c, 821
 ARMCM7_STM32H7/timer.c, 824
 HCS12/timer.c, 826
 timer.h, 1020
 TRICORE_TC2/timer.c, 828
 TRICORE_TC3/timer.c, 830
tios
 tTimerRegs, 78
total_size
 tFileEraseInfo, 66
TRICORE_TC2/can.c
 CanGetSpeedConfig, 153

CanInit, 154
CanReceivePacket, 154
canTiming, 155
CanTransmitPacket, 154
TRICORE_TC2/cpu.c
 CpuEnableUncorrectableBitErrorTrap, 219
 CpuInit, 220
 CpuMemcpy, 220
 CpuMemSet, 220
 CpuStartUserProgram, 221
TRICORE_TC2/flash.c
 blockInfo, 454
 bootBlockInfo, 454
 FlashAddToBlock, 447
 FlashDone, 448
 FlashErase, 448
 FlashEraseLogicalSector, 448
 FlashEraseSectors, 449
 FlashGetSectorIdx, 449
 Flash GetUserProgBaseAddress, 450
 FlashInit, 450
 FlashInitBlock, 450
 flashLayout, 454
 FlashSwitchBlock, 451
 FlashTranslateToNonCachedAddress, 451
 FlashVerifyChecksum, 452
 FlashWrite, 452
 FlashWriteBlock, 452
 FlashWriteChecksum, 453
 FlashWritePage, 453
TRICORE_TC2/flash.h
 FlashDone, 565
 FlashErase, 565
 Flash GetUserProgBaseAddress, 566
 FlashInit, 566
 FlashVerifyChecksum, 567
 FlashWrite, 567
 FlashWriteChecksum, 568
TRICORE_TC2/nvm.c
 NvmDone, 745
 NvmErase, 746
 Nvm GetUserProgBaseAddress, 746
 NvmInit, 746
 NvmVerifyChecksum, 747
 NvmWrite, 747
TRICORE_TC2/Tasking/cpu_comp.c
 CpuIrqDisable, 654
 CpuIrqEnable, 654
TRICORE_TC2/timer.c
 TimerGet, 827
 TimerInit, 828
 TimerReset, 828
 TimerUpdate, 828
TRICORE_TC2/types.h
 blt_addr, 885
 blt_bool, 885
 blt_char, 885
 blt_int16s, 885
 blt_int16u, 885
 blt_int32s, 886
 blt_int32u, 886
 blt_int8s, 886
 blt_int8u, 886
TRICORE_TC3/can.c
 CanGetSpeedConfig, 157
 CanInit, 157
 CanReceivePacket, 158
 canTiming, 159
 CanTransmitPacket, 158
TRICORE_TC3/cpu.c
 CpuEnableUncorrectableEccErrorTrap, 222
 CpuInit, 222
 CpuMemcpy, 222
 CpuMemSet, 223
 CpuStartUserProgram, 223
TRICORE_TC3/flash.c
 blockInfo, 464
 bootBlockInfo, 464
 FlashAddToBlock, 457
 FlashDone, 457
 FlashErase, 457
 FlashEraseLogicalSectors, 458
 FlashEraseSectors, 458
 FlashGetSectorIdx, 460
 Flash GetUserProgBaseAddress, 460
 FlashInit, 460
 FlashInitBlock, 461
 flashLayout, 464
 FlashSwitchBlock, 461
 FlashVerifyChecksum, 462
 FlashWrite, 462
 FlashWriteBlock, 462
 FlashWriteChecksum, 463
 FlashWritePage, 463
TRICORE_TC3/flash.h
 FlashDone, 569
 FlashErase, 569
 Flash GetUserProgBaseAddress, 570
 FlashInit, 570
 FlashVerifyChecksum, 571
 FlashWrite, 571
 FlashWriteChecksum, 572
TRICORE_TC3/nvm.c
 NvmDone, 748
 NvmErase, 749
 Nvm GetUserProgBaseAddress, 749
 NvmInit, 749
 NvmVerifyChecksum, 750
 NvmWrite, 750
TRICORE_TC3/Tasking/cpu_comp.c
 CpuIrqDisable, 655
 CpuIrqEnable, 655
TRICORE_TC3/timer.c
 TimerGet, 829
 TimerInit, 830
 TimerReset, 830

TimerUpdate, 830
TRICORE_TC3/types.h
 blt_addr, 887
 blt_bool, 887
 blt_char, 887
 blt_int16s, 887
 blt_int16u, 887
 blt_int32s, 888
 blt_int32u, 888
 blt_int8s, 888
 blt_int8u, 888
tscr1
 tTimerRegs, 78
tscr2
 tTimerRegs, 78
tseg1
 tCanBusTiming, 52
tseg2
 tCanBusTiming, 52
tSrecLineParseObject, 75
 address, 75
 data, 75
 line, 75
tSrecLineType
 file.h, 994
tstamp
 tCanRxMsgSlot, 60
 tCanTxMsgSlot, 61
tTimerRegs, 76
 cforc, 76
 oc7d, 76
 oc7m, 76
 tc, 77
 tcnt, 77
 tctl1, 77
 tctl2, 77
 tctl3, 77
 tctl4, 77
 tf1g1, 77
 tf1g2, 78
 tie, 78
 tios, 78
 tscr1, 78
 tscr2, 78
 ttov, 78
ttov
 tTimerRegs, 78
tud_suspend_cb
 ARMCM33_STM32H5/usb.c, 895
 ARMCM4_TM4C/usb.c, 945
 ARMCM4_XMC4/usb.c, 948
tXcpInfo, 79
 connected, 79
 ctoData, 79
 ctoLen, 79
 ctoPending, 80
 mta, 80
 protection, 80
s_n_k_resource, 80
txSlot
 tCanRegs, 59
Type definitions of a port, 25
types.h, 831, 833, 835, 837, 839, 842, 844, 846, 848, 851, 853, 855, 857, 859, 861, 863, 865, 867, 869, 872, 874, 876, 878, 880, 882, 884, 886
uip_tcp_appstate_t, 81
USB driver of a port, 25
usb.c, 888, 895, 898, 904, 912, 918, 924, 932, 938, 944, 947, 950, 958
usb.h, 1021
 UsbFree, 1022
 UsbInit, 1022
 UsbReceivePacket, 1022
 UsbTransmitPacket, 1023
UsbFifoMgrCreate
 _template/usb.c, 890
 ARMCM33_STM32L5/usb.c, 900
 ARMCM33_STM32U5/usb.c, 906
 ARMCM3_STM32F1/usb.c, 913
 ARMCM3_STM32F2/usb.c, 920
 ARMCM4_STM32F3/usb.c, 926
 ARMCM4_STM32F4/usb.c, 933
 ARMCM4_STM32L4/usb.c, 940
 ARMCM7_STM32F7/usb.c, 952
 ARMCM7_STM32H7/usb.c, 959
UsbFifoMgrInit
 _template/usb.c, 890
 ARMCM33_STM32L5/usb.c, 900
 ARMCM33_STM32U5/usb.c, 906
 ARMCM3_STM32F1/usb.c, 914
 ARMCM3_STM32F2/usb.c, 920
 ARMCM4_STM32F3/usb.c, 926
 ARMCM4_STM32F4/usb.c, 934
 ARMCM4_STM32L4/usb.c, 940
 ARMCM7_STM32F7/usb.c, 952
 ARMCM7_STM32H7/usb.c, 960
UsbFifoMgrRead
 _template/usb.c, 891
 ARMCM33_STM32L5/usb.c, 900
 ARMCM33_STM32U5/usb.c, 907
 ARMCM3_STM32F1/usb.c, 914
 ARMCM3_STM32F2/usb.c, 920
 ARMCM4_STM32F3/usb.c, 927
 ARMCM4_STM32F4/usb.c, 934
 ARMCM4_STM32L4/usb.c, 940
 ARMCM7_STM32F7/usb.c, 953
 ARMCM7_STM32H7/usb.c, 960
UsbFifoMgrScan
 _template/usb.c, 891
 ARMCM33_STM32L5/usb.c, 901
 ARMCM33_STM32U5/usb.c, 907
 ARMCM3_STM32F1/usb.c, 914
 ARMCM3_STM32F2/usb.c, 921
 ARMCM4_STM32F3/usb.c, 927
 ARMCM4_STM32F4/usb.c, 934
 ARMCM4_STM32L4/usb.c, 941

ARMCM7_STM32F7/usb.c, 953
ARMCM7_STM32H7/usb.c, 960
UsbFifoMgrWrite
 _template/usb.c, 891
 ARMCM33_STM32L5/usb.c, 901
 ARMCM33_STM32U5/usb.c, 908
 ARMCM3_STM32F1/usb.c, 915
 ARMCM3_STM32F2/usb.c, 921
 ARMCM4_STM32F3/usb.c, 928
 ARMCM4_STM32F4/usb.c, 935
 ARMCM4_STM32L4/usb.c, 941
 ARMCM7_STM32F7/usb.c, 954
 ARMCM7_STM32H7/usb.c, 961
UsbFree
 _template/usb.c, 892
 ARMCM33_STM32H5/usb.c, 896
 ARMCM33_STM32L5/usb.c, 902
 ARMCM33_STM32U5/usb.c, 908
 ARMCM3_STM32F1/usb.c, 915
 ARMCM3_STM32F2/usb.c, 922
 ARMCM4_STM32F3/usb.c, 928
 ARMCM4_STM32F4/usb.c, 935
 ARMCM4_STM32L4/usb.c, 942
 ARMCM4_TM4C/usb.c, 945
 ARMCM4_XMC4/usb.c, 948
 ARMCM7_STM32F7/usb.c, 954
 ARMCM7_STM32H7/usb.c, 961
 usb.h, 1022
UsbInit
 _template/usb.c, 892
 ARMCM33_STM32H5/usb.c, 896
 ARMCM33_STM32L5/usb.c, 902
 ARMCM33_STM32U5/usb.c, 908
 ARMCM3_STM32F1/usb.c, 915
 ARMCM3_STM32F2/usb.c, 922
 ARMCM4_STM32F3/usb.c, 928
 ARMCM4_STM32F4/usb.c, 935
 ARMCM4_STM32L4/usb.c, 942
 ARMCM4_TM4C/usb.c, 946
 ARMCM4_XMC4/usb.c, 949
 ARMCM7_STM32F7/usb.c, 954
 ARMCM7_STM32H7/usb.c, 961
 usb.h, 1022
UsbReceiveByte
 _template/usb.c, 892
 ARMCM33_STM32H5/usb.c, 896
 ARMCM33_STM32L5/usb.c, 902
 ARMCM33_STM32U5/usb.c, 908
 ARMCM3_STM32F1/usb.c, 916
 ARMCM3_STM32F2/usb.c, 922
 ARMCM4_STM32F3/usb.c, 928
 ARMCM4_STM32F4/usb.c, 936
 ARMCM4_STM32L4/usb.c, 942
 ARMCM4_TM4C/usb.c, 946
 ARMCM4_XMC4/usb.c, 949
 ARMCM7_STM32F7/usb.c, 954
 ARMCM7_STM32H7/usb.c, 962
 usb.h, 1022
UsbReceivePacket
 _template/usb.c, 893
 ARMCM33_STM32H5/usb.c, 897
 ARMCM33_STM32L5/usb.c, 902
 ARMCM33_STM32U5/usb.c, 909
 ARMCM3_STM32F1/usb.c, 916
 ARMCM3_STM32F2/usb.c, 922
 ARMCM4_STM32F3/usb.c, 929
 ARMCM4_STM32F4/usb.c, 936
 ARMCM4_STM32L4/usb.c, 942
 ARMCM4_TM4C/usb.c, 946
 ARMCM4_XMC4/usb.c, 949
 ARMCM7_STM32F7/usb.c, 955
 ARMCM7_STM32H7/usb.c, 962
 usb.h, 1022
UsbReceivePipeBulkOUT
 _template/usb.c, 893
 ARMCM33_STM32L5/usb.c, 903
 ARMCM33_STM32U5/usb.c, 909
 ARMCM3_STM32F1/usb.c, 916
 ARMCM3_STM32F2/usb.c, 923
 ARMCM4_STM32F3/usb.c, 929
 ARMCM4_STM32F4/usb.c, 936
 ARMCM4_STM32L4/usb.c, 943
 ARMCM7_STM32F7/usb.c, 955
 ARMCM7_STM32H7/usb.c, 962
UsbTransmitByte
 _template/usb.c, 893
 ARMCM33_STM32L5/usb.c, 903
 ARMCM33_STM32U5/usb.c, 909
 ARMCM3_STM32F1/usb.c, 917
 ARMCM3_STM32F2/usb.c, 923
 ARMCM4_STM32F3/usb.c, 929
 ARMCM4_STM32F4/usb.c, 937
 ARMCM4_STM32L4/usb.c, 943
 ARMCM7_STM32F7/usb.c, 955
 ARMCM7_STM32H7/usb.c, 963
UsbTransmitPacket
 _template/usb.c, 894
 ARMCM33_STM32H5/usb.c, 897
 ARMCM33_STM32L5/usb.c, 903
 ARMCM33_STM32U5/usb.c, 911
 ARMCM3_STM32F1/usb.c, 917
 ARMCM3_STM32F2/usb.c, 923
 ARMCM4_STM32F3/usb.c, 931
 ARMCM4_STM32F4/usb.c, 937
 ARMCM4_STM32L4/usb.c, 943
 ARMCM4_TM4C/usb.c, 947
 ARMCM4_XMC4/usb.c, 950
 ARMCM7_STM32F7/usb.c, 957
 ARMCM7_STM32H7/usb.c, 963
 usb.h, 1023
UsbTransmitPipeBulkIN
 _template/usb.c, 894
 ARMCM33_STM32L5/usb.c, 904
 ARMCM33_STM32U5/usb.c, 911
 ARMCM3_STM32F1/usb.c, 917
 ARMCM3_STM32F2/usb.c, 924
 ARMCM4_STM32F3/usb.c, 931

ARMCM4_STM32F4/usb.c, 937
ARMCM4_STM32L4/usb.c, 944
ARMCM7_STM32F7/usb.c, 957
ARMCM7_STM32H7/usb.c, 963

writeptr
 tFifoCtrl, 64

xcp.c, 1023
 XcpCmdBuildCheckSum, 1025
 XcpCmdConnect, 1026
 XcpCmdDisconnect, 1026
 XcpCmdGetId, 1026
 XcpCmdGetSeed, 1027
 XcpCmdGetStatus, 1027
 XcpCmdProgram, 1027
 XcpCmdProgramClear, 1028
 XcpCmdProgramMax, 1028
 XcpCmdProgramPrepare, 1029
 XcpCmdProgramReset, 1029
 XcpCmdProgramStart, 1029
 XcpCmdSetMta, 1030
 XcpCmdShortUpload, 1030
 XcpCmdSynch, 1030
 XcpCmdUnlock, 1031
 XcpCmdUpload, 1031
 XcpComputeChecksum, 1032
 XcpGetOrderedLong, 1032
 XcpGetSeed, 1032
 XcpInit, 1033
 XcpIsConnected, 1033
 XcpPacketReceived, 1033
 XcpPacketTransmitted, 1034
 XcpProtectResources, 1034
 XcpSetCtoError, 1034
 XcpSetOrderedLong, 1035
 XcpTransmitPacket, 1035
 XcpVerifyKey, 1035

xcp.h, 1036
 XCP_PACKET_RECEIVED_HOOK_EN, 1039
 XcpInit, 1040
 XcpIsConnected, 1040
 XcpPacketReceived, 1040
 XcpPacketTransmitted, 1041
 XCP_PACKET_RECEIVED_HOOK_EN
 xcp.h, 1039
 XcpCmdBuildCheckSum
 xcp.c, 1025
 XcpCmdConnect
 xcp.c, 1026
 XcpCmdDisconnect
 xcp.c, 1026
 XcpCmdGetId
 xcp.c, 1026
 XcpCmdGetSeed
 xcp.c, 1027
 XcpCmdGetStatus
 xcp.c, 1027
 XcpCmdProgram

 xcp.c, 1027
 XcpCmdProgramClear
 xcp.c, 1028
 XcpCmdProgramMax
 xcp.c, 1028
 XcpCmdProgramPrepare
 xcp.c, 1029
 XcpCmdProgramReset
 xcp.c, 1029
 XcpCmdProgramStart
 xcp.c, 1029
 XcpCmdSetMta
 xcp.c, 1030
 XcpCmdShortUpload
 xcp.c, 1030
 XcpCmdSynch
 xcp.c, 1030
 XcpCmdUnlock
 xcp.c, 1031
 XcpCmdUpload
 xcp.c, 1031
 XcpComputeChecksum
 xcp.c, 1032
 XcpGetOrderedLong
 xcp.c, 1032
 XcpGetSeed
 xcp.c, 1032
 XcpInit
 xcp.c, 1033
 xcp.h, 1040
 XcpIsConnected
 xcp.c, 1033
 xcp.h, 1040
 XcpPacketReceived
 xcp.c, 1033
 xcp.h, 1040
 XcpPacketTransmitted
 xcp.c, 1034
 xcp.h, 1041
 XcpProtectResources
 xcp.c, 1034
 XcpSetCtoError
 xcp.c, 1034
 XcpSetOrderedLong
 xcp.c, 1035
 XcpTransmitPacket
 xcp.c, 1035
 XcpVerifyKey
 xcp.c, 1035