Stack & Heap

Die großen Unbekannten der Embedded Software verstehen und beherrschen

Martin Gisbert, IAR Systems

Stack und Heap werden oft in einem Atemzug genannt, da es sich in beiden Fällen um nicht-statischen Speicher handelt. Eine weitere unangenehme Gemeinsamkeit ist der begrenzte Determinismus beim Zugriff und die Risiken bei Überläufen. Der folgende Artikel gibt eine Übersicht über die Funktionsweise des Stack und Tipps zur richtigen Dimensionierung. Da der Heap im Gegensatz zum obligatorischen Stack in Embedded Systemen relativ wenig eingesetzt wird, fällt die Beleuchtung dieses dynamischen Speichers kürzer aus.

Stack Übersicht

Der Laufzeit Stack, auch Stapelspeicher genannt, ist ein definierter Bereich im RAM Speicher, dessen Größe vom Anwender festgelegt wird. Der Linker reserviert diesen Bereich und platziert den Stack in der Regel im unteren Bereich des RAM oberhalb der globalen und statischen Variablen. Der Zugriff auf die Inhalte erfolgt über den Stack Pointer, der bei der Initialisierung auf das obere Ende des Stacks gesetzt wird, zur Laufzeit wächst der belegte Teil des Stack nach unten.

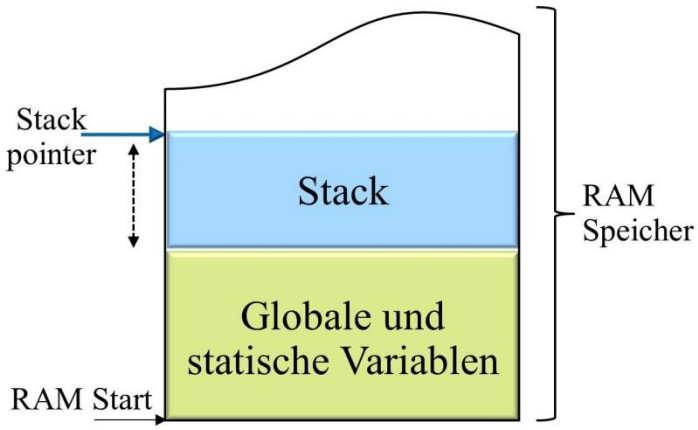

Abbildung 1: Anordnung des Laufzeit Stack

Auf dem Stack werden lokale Variablen einer Funktion gehalten, sofern sie nicht vom Compiler in Registern allokiert werden, also beispielsweise nicht skalare Variablen wie Arrays oder Strukturen. Sobald die Funktion beendet ist, werden diese Daten wieder vom Stack entfernt. Außerdem werden die Inhalte von Registern und Rücksprungadressen beim Aufruf von Unterfunktionen auf dem Stack abgelegt. Auch Funktionsparameter werden ggf. über den Stack übertragen.

Stack Überlauf
Durch die Angabe der Stackgröße wird dem Linker lediglich mitgeteilt, dass er diesen Bereich nicht anderweitig belegen darf. Der Mikroprozessor hat aber keinerlei Kenntnisse über die untere Grenze des Stacks und wird sie bedingungslos überschreiten, falls der Stackbedarf die Einschätzung des Entwicklers übersteigt.

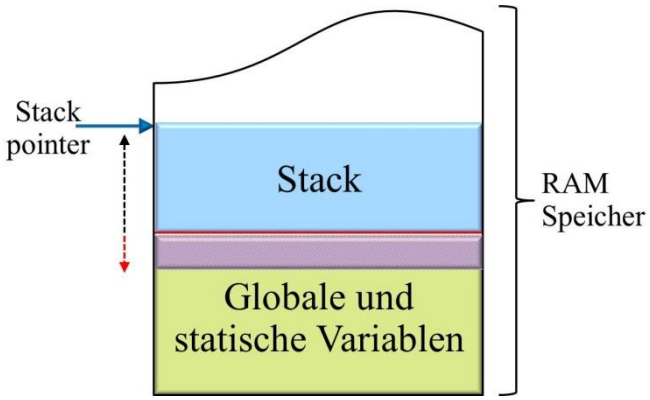

Abbildung 2: Stack Überlauf

Die Folge ist ein Überlauf des Stack, was zum einen bedeutet, dass globale Variablen mit zufälligen Daten überschrieben werden. Genauso gefährlich ist aber auch die Umkehrfolge, nämlich dass der Inhalt des Stack überschrieben wird, wenn die Variablen in der Konfliktzone beschrieben und damit etwa Rücksprungadressen aus Funktionen korrumpiert werden. Die Konsequenzen sind in jedem Fall eine Fehlfunktion oder gar ein Absturz der Anwendung. Das Ziel muss daher sein, den Stack groß genug zu wählen, damit Überläufe vermieden werden. Aber er soll auch nicht zu groß sein, da hierdurch RAM Speicher verschwendet würde.

Methoden zur korrekten Dimensionierung des Stack
Zur Ermittlung der benötigten Stack Größe gibt es einige hilfreiche Methoden. Zunächst lässt sich die Tatsache ausnutzen, dass der Compiler den Stackbedarf jeder Funktion kennt und diese im Listfile dokumentiert. Mit den passenden Tools wie beispielsweise der IAR Embedded Workbench lässt sich damit eine Stack-Analyse durchführen und ein erster Richtwert berechnen (Abbildung 1).

```
******************** STACK USAGE ********************

  Call Graph Root Category  Max Use  Total Use
  ------------------------  -------  ---------
  interrupt                       8          8
  main_root                     164        164

  Maximum call chain            164 bytes

    "__iar_program_start"         0
    "__cmain"                     0
    "main"                        8
    "DrawPicture"                24
...
```

Abbildung 3: Ergebnis der Stack Analyse

Indirekte Funktionsaufrufe oder Rekursionen können dabei berücksichtigt werden, nicht aber der Bedarf für gerettete Register. Daher sollte der wirkliche Bedarf noch empirisch verifiziert werden. Wird beim Debuggen der Stackbereich mit einem bestimmten Muster gefüllt, lässt sich nach einem vollständigen Testdurchlauf feststellen, wie viel vom Stack noch unberührt ist. Moderne Debugger bieten hierfür eine grafische Unterstützung an. Abbildung 4 zeigt ein Beispiel des C-SPY Debuggers der IAR Embedded Workbench, bei dem der Anteil der „verbrauchten“ Stacks angezeigt wird:

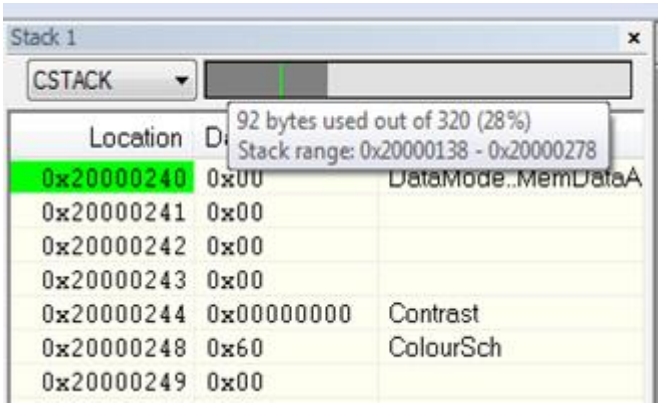

Abbildung 4: Stack Ansicht des Debuggers

Eine weitere Möglichkeit, die maximale Stack-Tiefe während der Laufzeit zu messen, ist, den Stack Pointer regelmäßig, z.B. in einer Timer Interrupt Routine, zu pollen.

```
char *highStack, *lowStack;

int main(int argc, char *argv[])
{
    highStack = (char *)&argc;
    lowStack = highStack;
    . . .


void sampling_timer_interrupt_handler(void)
{
     char* currentStack;
     int a;
     currentStack = (char *)&a;
     if (currentStack < lowStack)
            lowStack = currentStack;
}
```

Abbildung 5: Pollen des Stack Pointers

Im Beispiel in Abbildung 5 wird zu Beginn des Programms, also bei leerem Stack, die obere Grenze des Stacks in der Variable highStack gespeichert. Bei jedem Aufruf der Polling Routine wird die aktuelle Position des Stack Pointers ausgelesen und der Wert der aktuellen Stack Tiefe erfasst. Am Ende des Tests ist highStack – lowStack der maximale Stack Bedarf, vorausgesetzt, das Pollen wurde schnell genug ausgeführt.

Um bei einem trotz allem auftretenden Überlauf des Stacks die möglicherweise fatalen Folgen zu vermeiden, lässt sich zwischen den Speicherbereichen von Stack und Variablen eine Sicherheitszone platzieren (siehe Abbildung 6). Diese Sicherheitszone wird mit einem bestimmten Muster initialisiert, das während des Laufes regelmäßig überprüft wird. Wurde das Muster überschrieben, ist der Stack zwar übergelaufen aber noch keine Daten korrumpiert, vorausgesetzt, die Sicherheitszone ist ausreichend groß.

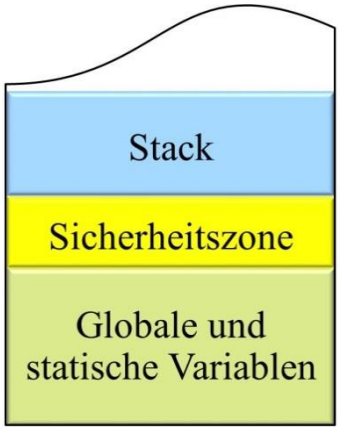

Abbildung 6: Sicherheitszone im Speicher anlegen

Der Vorteil dieser Methode ist, dass sie auch außerhalb des Labors verwendet werden kann. Wird während des Betriebes durch regelmäßige Tests ein Schreibzugriff auf die Sicherheitszone erkannt, kann beispielsweise das Gerät kontrolliert heruntergefahren werden oder zumindest durch eine Anzeige der Stack Überlauf bekannt gegeben werden. Verfügt der verwendete Mikrocontroller über eine Memory Protection Unit, kann diese zur Überwachung der Sicherheitszone verwendet werden. In der Debugging Umgebung lässt sich ein Zugriff mithilfe eines Data Breakpoint sofort feststellen.

Dynamische Speicherverwaltung mit dem Heap

Der Heap ist, ähnlich dem Stack, ein dedizierter Teil des RAM, in dem von der Applikation Speicher dynamisch belegt werden kann. Die wichtigsten Funktionen hierfür sind `malloc` und `free`: Mit `malloc` wird ein Block einer bestimmten Größe auf dem Heap allokiert und die Adresse dieses Blocks zurückgegeben. Varianten sind `calloc` (Initialisierung des allokierten Speichers mit 0) und `realloc` (Vergrößerung oder Verkleinerung eines bereits allokierten Blocks). Wird der Speicher nicht mehr benötigt, wird er mit der `free` Funktion wieder frei gegeben. Kann der angeforderte Speicher nicht zur Verfügung gestellt werden, returniert `malloc` den Wert NULL.

```
// Allokieren von dynamischem Speicher für
// die der Struktur s1 im Heap
struct s *s1;
s1 = malloc (sizeof (struct s));
// Rückgabe dynamisch allokierten Speichers
free (s1);
```

Abbildung 7: generelle Verwendung von `malloc` und `free`

Die genaue Funktion von `malloc` u.a. ist nicht im ANSI-C Standard definiert. Die Implementierung des Heap Handlers bleibt damit dem Anwender überlassen, falls er nicht universelle Ausführungen wie dlmalloc verwenden möchte.

Im Gegensatz zum Stack ist der Heap für Embedded Systeme nicht obligatorisch und wird sehr selten verwendet, nicht zuletzt wegen des nicht-deterministischen Verhaltens: Wenn die dynamisch allokierten Objekte unterschiedliche Größen haben, wird der Heap mit der Zeit fragmentiert. Das Allokieren von Speicher dauert damit immer länger und ist möglicherweise irgendwann gar nicht mehr möglich. Außerdem lässt sich der Heap sehr leicht korrumpieren, z.B. durch das mehrmalige Freigeben eines Speicherbereiches oder den Zugriff außerhalb der Grenzen eines allokierten Objektes. Da der Heap keine linearer Speicher ist, sondern auch Zeiger auf weitere Blöcke enthält, kann der dynamische Speicher dadurch komplett zerstört werden und ist nur durch einen Neustart wieder herzustellen.

Die meisten Heap Fehler lassen sich durch den Einsatz eines „Wrappers“ erkennen. Hierbei wird beim Allokieren von Speicher vor und hinter dem eigentlichen Block ein weiteres Feld eingefügt, mit dem die Konsistenz des Heap überwacht wird:

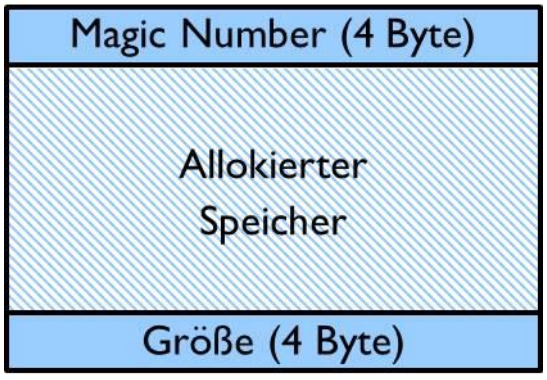

Abbildung 8: Heap Wrapper

Für den Wrapper werden zwei weitere Heap Funktionen, z.B. `MyMalloc` und `MyFree` implementiert. `MyMalloc` ruft `malloc` auf, allokiert aber 8 Byte mehr, die mit einem bestimmten Muster sowie der Größe des Blocks initialisiert werden. `MyFree` überprüft vor dem Aufruf von `free`, ob das Füllmuster noch erhalten und bricht andernfalls mit einer Fehlermeldung ab.

Zusammenfassung
Der Laufzeit-Stack tritt oft erst dann in Erscheinung, wenn er überläuft und das Programm nicht mehr korrekt läuft oder gar abstürzt. Daher muss generell sicher sichergestellt werden, dass der Stapelspeicher für alle Anwendungsfälle ausreichend groß dimensioniert wurde um spätere Überraschungen mit möglicherweise fatalen Folgen zu verhindern. Auf die Verwendung des dynamischen Speichers sollte in Embedded Systemen nach Möglichkeit ganz verzichtet werden. Die Implementierung eines Wrappers kann jedoch hilfreich sein, um einen korrumpierten Heap zumindest zu erkennen.

Autor
Martin Gisbert ist seit mehr als 15 Jahren in unterschiedlichen Bereichen der Embedded Systems Branche tätig. Seit 2009 arbeitet er als Field Application Engineer bei IAR Systems in München und ist für Support, Schulung und technisches Marketing von IAR Produkten zuständig.

.

Kontakt
Internet: www.iar.com
Email: martin.gisbert@iar.com