Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik

Informationstechnik für Luft- und Raumfahrt
Lehrstuhl für Informatik 8
Prof. Dr. Sergio Montenegro

Bachelorarbeit

Filter-Konzept für diagnostische Reports

Vorgelegt von
Andreas Schartel
Matr.-Nr.: 1807554

Angefertigt beim
Deutschen Zentrum für Luft- und Raumfahrt (DLR) Braunschweig
Einrichtung für Simulations- und Softwaretechnik

Prüfer:	Prof. Dr. Sergio Montenegro
Betreuer (DLR):	Dr. rer. nat. Olaf Maibaum

Würzburg, 29.07.2014

Aufgabenstellung

Essentiell für den Betrieb von Satelliten ist die Erfassung von Daten zur Überwachung der Funktionen des Satelliten. Zu diesem Zweck definiert die ECSS-E-70-41A den Service "Housekeeping and Diagnostic Report". Momentan wird in dem Kompaktsatellitenprogramm des DLR dieser Service lediglich zur Generierung von Housekeeping-Datenpaketen eingesetzt. Die im Standard definierten Möglichkeiten zum Filtern von Daten werden nicht genutzt.

In der Bachelorarbeit soll zu diesem Zweck ein Filter-Konzept ausgearbeitet werden, welches auf dem Tasking-Framework aufsetzt. Das Tasking-Framework ist ein zentraler Bestandteil im Projekt OBC-NG (Onboard Computer – Next Generation) und dient dem Aufbau eines reaktiven Onboard-Systems. Kernelement des Tasking-Frameworks ist der Datenaustausch zwischen einzelnen Rechenprozessen einer Applikation. Das zu entwickelnde Filter-Konzept soll sich in diesen internen Datenaustausch einbinden, um Teile der Daten entsprechend der Filtereinstellung als Telemetry-Daten weiter zu leiten. Die in der Bachelorarbeit zu bearbeitenden Aufgaben sind:

- Literaturrecherche im Hinblick auf Filter-Konzepte für Onboard-Systeme in Raumfahrtanwendungen
- Erstellen eines Filter-Konzepts
- Richtlinien für den Datenaustausch bei Nutzung von Filtern
- Umsetzung des Filter-Konzepts als Prototyp
- Bewertung des Konzepts hinsichtlich Speicherplatz und Rechenzeit
- Dokumentation der Ergebnisse in Form der Bachelorarbeit

Eine volle Kompatibilität zum PUS-Service „Request and Reports“ ist im Rahmen der Arbeit nicht gefordert.

Zusammenfassung

Diese Arbeit beschreibt den Entwurf eines Konzepts zur Filterung diagnostischer Reports sowie die Evaluation des Konzepts anhand eines Prototyps. Die Funktionsweise des Filter-Konzepts soll sich dabei an der im ECSS Standard E-70-41A beschriebenen Möglichkeit zur Filterung von Reports orientieren. Als Grundlage für den Entwurf diente das beim DLR entwickelte Tasking Framework sowie Code-Ausschnitte des diagnostischen Report Systems aus der aktuellen Eu:Cropis Mission. Der Prototyp wurde in C++ unter einem 32-Bit Linux System programmiert und getestet. Neben dem Entwurf selbst, wurde bei der Arbeit besonders Wert darauf gelegt die in der Raumfahrt üblichen Richtlinien zur Sicherung der Softwarequalität einzuhalten. Abschließende Tests validierten nicht nur die korrekte Funktionsweise des Prototyps, sondern ermöglichten auch eine Bewertung der Leistungsfähigkeit des Konzepts hinsichtlich des benötigten Speicherplatzes und Rechenzeit.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

AOCS ..... Attitude and Orbit Control System
APID ..... Applikations ID
CCSDS ..... Consultative Commitee for Space Data Systems
CPU ..... Central Processing Unit

DLR	Deutsches Zentrum für Luft- und Raumfahrt
ECSS	European Cooperation for Space Standardization
ESA	European Space Agency
FIFO	First In, First Out (Datenstruktur)
MISRA	Motor Industry Software Reliability Association
NASA	National Aeronautics and Space Administration
NFA	Number of Fixed-Length Arrays
OBC	Onboard Computer
OBC-NG	Onboard Computer - Next Generation
OBCP	Onboard Control Procedures
OBSW	Onboard Software
OBSW DP	Onboard Sotware Data Pool
PL Hdl.	Payload Handler
PUS	Packet Utilisation Standard
RAM	Random Access Memory
ROM	Read-Only Memory
RTOS	Real Time Operating System, Echtzeitbetriebssystem
RTTI	Run-Time Type Information
RWL Hdl.	Reaction Wheel Handler
SBC	Single Board Computer, Einplatinenrechner
SID	Structure ID, Identifiziert eine Report Definition
STR Hdl.	Star Tracker Handler
TC	Telekommando
TDD	Test-Driven Development
TM	Telemetrie

1 Einleitung

Bei jeder Raumfahrtmission, unabhängig ihrer Größe und ihres Budgets, spielt die Telemetrie eine zentrale Rolle. Da jede Mission in irgendeiner Form entweder dem Informationsgewinn oder Informationstransport dient, ist es für den Erfolg der Mission unerlässlich Daten zur Erde übertragen zu können. Dabei wird grundsätzlich zwischen zwei Arten von Telemetrie unterschieden: Der *Nutzlast-* und der *Housekeeping-Telemetrie*. Mithilfe der Nutzlast-Telemetrie werden, wie der Name bereits suggeriert, die Daten der Nutzlast zur Erde übertragen also beispielsweise Messungen wissenschaftlicher Instrumente, Fotoaufnahmen oder Fernsehsignale. Art und Aufbau dieser Telemetriedaten sind dabei stark vom Missionsziel abhängig.
Die Housekeeping Telemetrie enthält dagegen alle Informationen, die zum Betrieb des Raumfahrzeugs nötig sind wie beispielsweise aktuelle Statusinformationen (Ausrichtung, Betriebsmodus, Temperatur), statistische Informationen (mittlere Temperatur, maximale Ladung der Batterie) oder Informationen über aufgetretene Ereignisse. Die Housekeeping Telemetriedaten dienen gewissermaßen dem Überleben des Raumfahrzeugs und sollten daher so ausgelegt sein, dass sie den Zustand des Raumfahrzeugs vollständig und unmissverständlich widerspiegeln.
Bei der Akquirierung der zu sendenden Statusinformationen gibt es dabei verschiedene Vorgehensweisen: Im einfachsten Fall wird eine Menge von zuvor festgelegten Parametern in einem bestimmten Intervall abgefragt und gesendet beziehungsweise, im Falle einer fehlenden Verbindung zur Bodenstation, zwischengespeichert. Dies hat jedoch den Nachteil, dass unter Umständen eine große Menge an Speicher und Bandbreite verschwendet wird, da für die vollständige Abbildung des Zustands des Raumfahrzeugs oft viele Parameter nötig sind, die sich selten bis nie ändern. Daher wäre es sinnvoll bestimmte Parameter oder Parametergruppen mit einem Filter versehen zu können, so dass sie nur gesendet werden, wenn sie sich nennenswert geändert haben.
Tatsächlich ist im ECSS Standard E-70-41A, in dem Empfehlungen für die Interaktion zwischen Satelliten und Bodenstation auf Applikationsebene beschrieben sind, solch ein gefilterter Housekeeping Service vorgesehen. Im Kompaktsatellitenprogramm des Deutschen Zentrums für Luft- und Raumfahrt (DLR) werden derzeit jedoch nur die Minimalanforderungen des ECSS E-70-41A Standards an den Housekeeping Service umgesetzt, die Möglichkeit zum Filtern der Daten wird nicht genutzt.
Kern dieser Arbeit soll daher sein, aufbauend auf dem im DLR verwendeten Tasking Framework sowie einer bereits bestehenden Implementierung des Housekeeping Services ein Konzept zur Generierung gefilterter Housekeeping Reports gemäß ECSS E-70-41A zu erarbeiten. Dieses Konzept soll dann anhand eines Prototypen getestet und hinsichtlich seiner Leistungsfähigkeit bewertet werden, wobei insbesondere der Speicherbedarf und die Laufzeit im Vordergrund stehen. Des Weiteren wird sowohl beim Entwurf als auch bei der Implementierung des Prototyps Wert darauf gelegt, die in der Raumfahrt

üblichen Standards und Empfehlungen bezüglich Softwarequalität einzuhalten, um die spätere Umsetzung des Konzepts in realen Systemen zu vereinfachen.

2 Grundlagen

2.1 Onboard Computer und Data Handling

2.1.1 Aufbau des Onboard Computers

Der Onboard Computer ist die Einheit in einem Raumfahrzeug, die für die Ausführung der Flugsoftware zuständig ist. Häufig wird fälschlicherweise angenommen der Onboard Computer wäre für Lageregelung, Telemetrie, Telekommando, Navigation, Kommandierung der Payload usw. verantwortlich. Tatsächlich sind dies jedoch alles Aufgaben der Onboard Software – der Onboard Computer dient lediglich deren Ausführung.
Der Aufbau des Onboard Computers kann je nach Art und Größe des Satelliten völlig unterschiedlich sein. Außerdem ist es nicht immer eindeutig, welche Komponenten dem Onboard Computer zugeordnet werden sollen und welche als Peripherie zu werten sind. Viele der heutzutage verwendeten Sensoren und Aktuatoren besitzen eigene Prozessoren bzw. Microcontroller, die teilweise über erhebliche Rechenleistung verfügen.
Generell kann man sagen, der Onboard Computer besteht aus einem oder mehreren Rechenknoten, die meistens als sogenannte *Single Board Computer* (SBC) ausgelegt sind. Bei Single Board Computern sind neben der CPU bereits alle für den Betrieb nötigen Komponenten wie etwa Boot-/Programmspeicher (ROM), Arbeitsspeicher (RAM) sowie je nach Bedarf Eingabe/Ausgabe Geräte und Netzwerkschnittstellen auf einer einzigen Platine enthalten (siehe Abbildung 1).

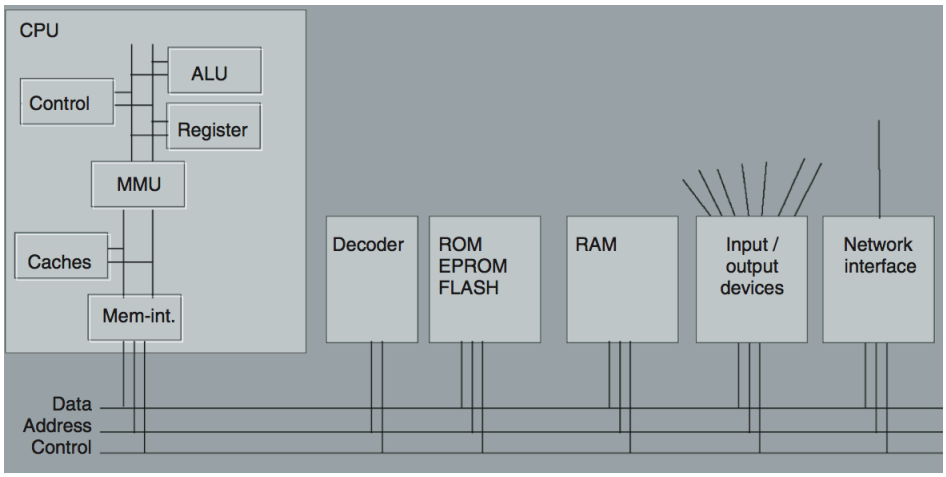


Abb. 1: Blockdiagramm eines Rechenknotens [1]

[1] Montenegro, [LWH09] [S.368, Figure 4.6.6]

Des Weiteren enthält ein Onboard Computer typischerweise noch Komponenten für die Stromversorgung, die (De-)Kodierung der Hochfrequenzsignale, Rekonfiguration und Neuprogrammierung sowie einen Massenspeicher. Diese Komponenten sind häufig als eigene Module ausgelegt, können unter Umständen jedoch auch direkt als Teil eines Rechenknotens auf dem SBC untergebracht sein (z.B. bei extrem kleinen Satelliten wie Cube Sats).

Da im Weltall eine deutlich höhere Strahlungsbelastung als auf der Erde herrscht, werden für den Onboard Computer meist spezielle Prozessoren verwendet, die eine extrem hohe Resistenz gegenüber elektromagnetischer, mechanischer und thermischer Belastung aufweisen. Im europäischen Raum sind insbesondere Prozessoren auf Basis der ERC32 und LEON Architektur beliebt, während in den USA häufig Prozessoren der PowerPC 603/750 Reihe verwendet werden.[2] Ein weiteres wichtiges Merkmal, das diesen Prozessoren, neben ihrer Robustheit, zum Erfolg in der Raumfahrt verholfen hat, ist die Tatsache, dass sie sogenannte *Echtzeitbetriebssysteme* (siehe Abschnitt 2.1.2) wie beispielsweise RTEMS oder das beim DLR entwickelte RODOS unterstützen. Allerdings besitzen strahlungsresistente Prozessoren auch den Nachteil, dass sie gegenüber handelsüblichen Prozessoren wesentlich langsamer sind. Moderne strahlungsresistente Prozessoren erreichen typischerweise nur eine Taktfrequenz von 20 bis 66 MHz[3] wohingegen heutzutage bereits viele Smartphone-Prozessoren mit 1GHz und mehr takten. Aus diesem Grund forscht das DLR derzeit im Rahmen des Projekts OBC-NG (On Board Computer - Next Generation), das auch in der Aufgabenstellung bereits erwähnt wurde, an einem Onboard Computer mit verschieden Rechenknoten aus strahlungsresistenter Hardware für kritische und handelsüblicher Hardware für weniger kritische, dafür aber leistungsinstensivere Aufgaben. Da sich solch leistungsfähige Systeme jedoch noch in der Entwicklung und Erprobung befinden muss für den Entwurf des Filter-Konzepts von einem herkömmlichen, langsamen OBC mit beschränkten Ressourcen ausgegangen werden.

2.1.2 Software

Die Software kann, wie auch zuvor schon die Hardware, von Satellit zu Satellit sehr unterschiedlich aufgebaut sein, doch auch hier gibt es häufig Gemeinsamkeiten und typische Architekturen.

Da die Software für den Betrieb von Satelliten immer komplexer wird, wird heutzutage bei nahezu allen Satelliten ein Echtzeitbetriebssystem (engl. Real Time Operating System, RTOS) verwendet, anstatt direkt auf der Hardware zu programmieren. Der Zuwachs an Komplexität liegt zum einen daran, dass Raumfahrzeuge immer mehr Auf-

[2] Eickhoff, vgl. [Eic11] [S.54]
[3] Eickhoff, vgl. [Eic11] [S.5]

gaben autonom, also ohne Eingriff vom Boden aus erledigen sollen, zum anderen werden auch immer mehr Komponenten, die früher als eigene Hardware in Form einer analogen oder digitalen Schaltung realisiert waren, durch Softwaremodule nachgebildet und ersetzt. Dies hat den Vorteil, dass sich Software auch im Orbit noch leicht korrigieren und verbessern lässt, erfordert im Gegenzug jedoch ein hohes Maß an Koordination innerhalb der Onboard Software, damit sich die einzelnen Module der Software nicht gegenseitig behindern. Hier kommt das RTOS ins Spiel: Ein RTOS dient, wie alle anderen Betriebssysteme auch, der Abstraktion und Verwaltung von Hardwareressourcen, es koordiniert also die Zugriffe der einzelnen Prozesse auf die Hardware. Ein RTOS muss im Gegensatz zu herkömmlichen Betriebssystemen jedoch in erster Linie deterministisch und zeitgesteuert sein, das heißt es muss für jeden beliebigen Zeitpunkt bestimmbar sein in welchem Zustand sich das System befand oder befinden wird. Außerdem muss der gewünschte Zeitpunkt der Ausführung einer Aktion möglichst exakt eingehalten werden können, um das Raumfahrzeug präzise zu steuern. Herkömmliche Betriebssysteme, die auf hohen Datendurchsatz optimiert sind, bieten diese Eigenschaften nicht und sind somit für den Einsatz in Raumfahrzeugen eher ungeeignet. Da Echtzeitbetriebssysteme in der Regel auf eingebetteten Systemen laufen, ergibt sich darüber hinaus noch die Anforderung, dass sie besonders ressourcenschonend hinsichtlich Rechenleistung und Speicher sein müssen. Neben diesen Anforderungen ist bei der Auswahl eines Betriebssystems für Raumfahrtanwendungen darauf zu achten, welche Software und Frameworks für dieses System verfügbar sind. Wünschenswert sind zum Beispiel Betriebssysteme, die bereits Funktionen wie Interprozesskommunikation, Standard-Netzwerkschnittstellen, Fehlererkennung und Ähnliches bieten.

Auf dem Echtzeitbetriebssystem aufsetzend wird dann die eigentliche Onboard Software des Raumfahrzeugs implementiert. Eine mögliche Architektur, die in dem Buch „Onboard Computers, Onboard Software and Satellite Operations: An Introduction“[4] von Jens Eickhoff vorgestellt wird, ist in Abbildung 2 dargestellt. Der Bootloader, der sich an einer fest definierten Adresse im Programmspeicher befindet, ist dafür zuständig das Betriebssystem zu starten und zu initialisieren. Das Betriebssystem lädt und initialisiert wiederum die Treiber zur Ansteuerung der I/O (Eingabe/Ausgabe) Schnittstellen des Microcontrollers (blaue Boxen). Diese Treiber können, je nach Microcontroller, entweder vollständig in Software oder teilweise auch in Hardware implementiert sein. Darüber hinaus gibt es häufig sogenannte *Equipment Handler* (gelbe Boxen), die der Steuerung und Kommandierung externer Geräte dienen. Typischerweise gibt es für jede Geräteklasse einen eigenen Handler also z.B. ein Reaction Wheel Handler (RWL Hdl.), Star Tracker Handler (STR Hdl.), Payload Handler (PL Hdl.) und so weiter. Diese Handler beziehen die Informationen, wie die Aktuatoren zu steuern sind, aus einem gemeinsamen Daten-Pool (OBSW DP) und legen im Gegenzug die von Senso-

[4] siehe [Eic11]

ren erhaltenen Daten, wie z.B. Lageinformationen des Star Trackers, im Daten-Pool ab. Sie bilden also eine Software Schnittstelle auf niedriger Ebene, enthalten dabei jedoch keinerlei übergeordnete Logik zur Regelung, Steuerung oder Automatisierung des Raumfahrzeugs. Der zuvor erwähnte Daten-Pool ist als eine Modellvorstellung eines gemeinsam genutzten Speicherbereichs zu verstehen. In der Regel handelt es sich dabei um Teile des Arbeitsspeichers, auf die mittels diverser Variablen und Pointern zugegriffen wird. Die eigentliche Regelung und Steuerung des Raumfahrzeugs geschieht in der sogenannten Applikationsebene, in der Aufgaben wie Lageregelung, Payload-Control und Power-Management implementiert werden. Jede dieser Funktionen wird dabei durch eine eigene Applikation repräsentiert. Die Applikationen tauschen über den Daten-Pool sowohl untereinander, als auch mit den Equipment Handlern Informationen aus. So kann beispielsweise die AOCS Applikation (Lageregelung) die Lageinformationen, die verschiedene Sensor Handler dort abgelegt haben aus dem Daten-Pool abrufen, diese Daten weiter verarbeiten und die berechneten Informationen zur Lagekorrektur wieder an den Daten-Pool ausgeben, von wo aus sie etwa vom Reaction Wheel Handler oder einem Magnet Torquer Handler abgerufen und in entsprechende Steuersignale für die Aktuatoren gewandelt werden können. Nicht dargestellt ist in Abbildung 2, wie die Interaktion zwischen Telemetrie Encoder bzw. Telekommando Decoder und dem Rest der Onboard Software abläuft. Auch dies kann auf unterschiedliche Art und Weise erfolgen. Die in Europa weit verbreiteten Empfehlungen der ECSS sehen im sogenannten *Packet Utilization Standard* (PUS) einen servicebasierten Ansatz vor, der in Abschnitt 2.2.2 näher erläutert wird.

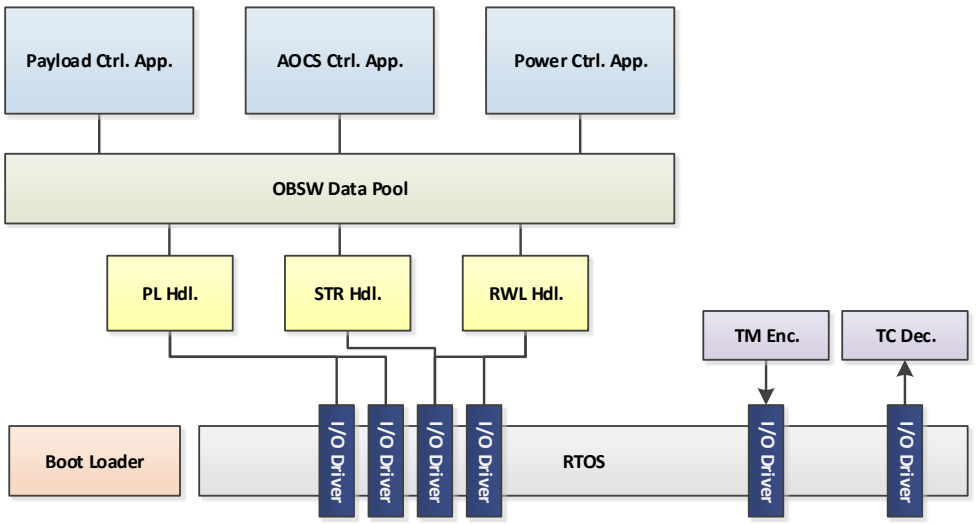


Abb. 2: Statische Architektur der OBSW[5]

[5]Zeichnung nach Eickhoff, [Eic11] [S.101, Figure 8.12]

2.2 Zusammenspiel von Bodenstationen und Satelliten

2.2.1 Übertragungsstrecke

Die Kommunikation zwischen Bodenstation und Raumfahrzeug geschieht heutzutage noch ausnahmslos über Radiowellen. Zwar gibt es erste Konzepte und Versuche zur Kommunikation über optische Kanäle, allerdings sind diese Systeme noch nicht marktreif. Häufig besitzen Raumfahrzeuge zwei unabhängige Übertragungsstrecken in zwei verschiedenen Frequenzbändern, da hohe Frequenzen zwar die Bandbreite erhöhnen, jedoch auch eine genauere Ausrichtung der Antennen erfordern. Durch den Einsatz einer Übertragungsstrecke mit geringer Frequenz für Telemetrie/Telekommando und einer mit hoher Frequenz für die Payload Daten kann sichergestellt werden, dass das Raumfahrzeug auch bei ungenauer Ausrichtung erreichbar bleibt und trotzdem, im Falle einer präzisen Ausrichtung, eine große Menge an Nutzlastdaten übertragen kann. Aus diesem Grund ist es auch nicht sinnvoll die Frequenz des Housekeeping Kanals zugunsten einer höheren Bandbreite zu erhöhen, weshalb eine Reduktion der zu sendenden Daten durch einen Filter umso wichtiger wird.

Die Übertragung der Daten erfolgt nach einem, vom *Consultative Committee for Space Data Systems* (CCSDS) herausgegebenen, standardisierten Protokoll, wodurch eine missions- und länderübergreifende Nutzung der Bodenstationen ermöglicht wird. Das Protokoll sieht dabei einen paketbasierten Ansatz vor, der in den Abbildungen 2 und 3 exemplarisch für den Downlink, also die Übertragung der Telemetrie vom Raumfahrzeug zur Bodenstation, dargestellt ist.

Dazu wurden im CCSDS Standard zwei verschiedene Datenstrukturen definiert: Sogenannte „Source Packets“ und „Transfer Frames“. Die Source Packets werden direkt von der Applikation, die etwas übertragen möchte, generiert. Sie können sowohl zu beliebigen Zeitpunkten als auch mit beliebiger Länge erstellt werden. Abgesehen von einem fest definierten Header kann auch der Inhalt der Pakete völlig beliebig sein und an die Anforderungen der jeweiligen Applikation angepasst werden. Der Header enthält neben einigen anderen Informationen wie Versionsnummer und Länge des Pakets eine eindeutige Applikations-ID (APID oder AP), mit deren Hilfe die Quelle eines Pakets identifiziert und das Paket entsprechend weitergeleitet werden kann. Diese Source Packets werden dann für die Übertragung auf die bereits zuvor genannten Transfer Frames aufgeteilt. Transfer Frames besitzen im Gegensatz zu Source Packets eine feste Länge und werden taktgesteuert generiert. Durch ihre Eigenschaften ermöglichen sie, trotz der störanfälligen Übertragungsstrecke, eine relativ stabile Verbindung. Da ein Source Packet an jeder beliebigen Stelle innerhalb eines Frames beginnen und enden kann, kann es sowohl passieren, dass mehrere Source Packets in einem Frame enthalten sind als auch dass ein Source Packet über mehrere Frames aufgeteilt wird. Darüber

hinaus sieht das CCSDS Protokoll vor, die physikalische Übertragunsstrecke mithilfe virtueller Kanalisierung aufzuteilen, um den Datenfluss verschiedener Quellen regulieren zu können. Somit könnten beispielsweise die Datenströme der Nutzlast von denen des Satellitenbuses oder aufgezeichnete Daten von Echtzeit-Daten getrennt und unterschiedlich priorisiert werden. Zu diesem Zweck besitzen die Transfer Frames eine eindeutige Zugehörigkeit zu einem der bis zu acht vorgesehenen virtuellen Kanäle. Um die virtuellen Kanäle übertragen zu können, müssen die Datenströme zuvor wieder zu einem einzigen „Master Channel“ zusammen gefasst und auf ein Trägersignal aufmoduliert werden.[6]

Die Aufteilung der Source Packets in Transfer Frames, das Multiplexen der verschiedenen Kanäle sowie die Modulierung des Signals geschehen in der Regel auf einem eigenen „CCSDS-Prozessor-Board“[7]. Die Onboard Software muss lediglich, wie bereits zuvor erwähnt, die Source Packets generieren und an das Telemetrie Interface weitergeben. Inhalt, Aufbau und Größe der Source Packets wurden dabei bewusst nicht spezifiziert, um das Übertragungsprotokoll möglichst flexibel zu halten. Das CCSDS Protokoll kann also gewissermaßen als ein standarisierter „Rahmen“ gesehen werden, in dem beliebige Daten übertragen werden können.

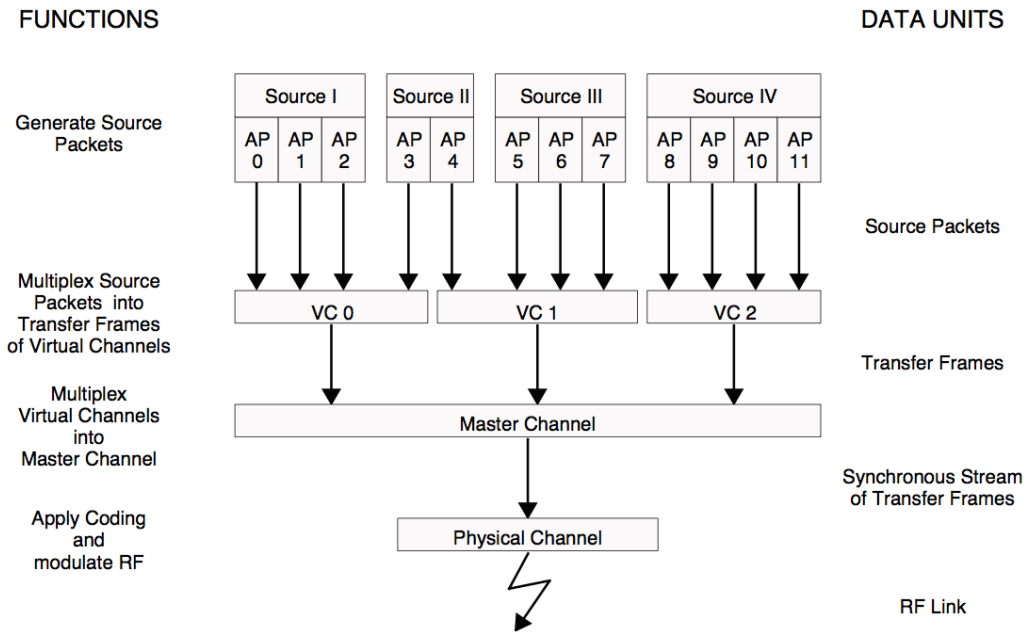


Abb. 3: Datenfluss der Telemetrie vom Raumfahrzeug zum physikalischen Kanal[8]

[6] CCSDS: Packet Telemetry, [ccs95]
[7] Eickhoff, vgl. [Eic11] [S.63]
[8] CCSDS: Packet Telemetry, [ccs95] [S. 2-4, Figure 2.2]

Auf Seiten der Bodenstation läuft ein ähnlicher Prozess ab, jedoch in umgekehrter Reihenfolge: Zuerst werden die Daten empfangen und demoduliert und anschließend die virtuellen Kanäle aus dem Master Channel herausgefiltert. Zum Schluss werden die Source Packets aus den unterschiedlichen Transfer Frames rekonstruiert und in Abhängigkeit ihrer Applikations-ID an verschiedene Senken weitergeleitet, wo die Daten analysiert, visualisiert oder archiviert werden.

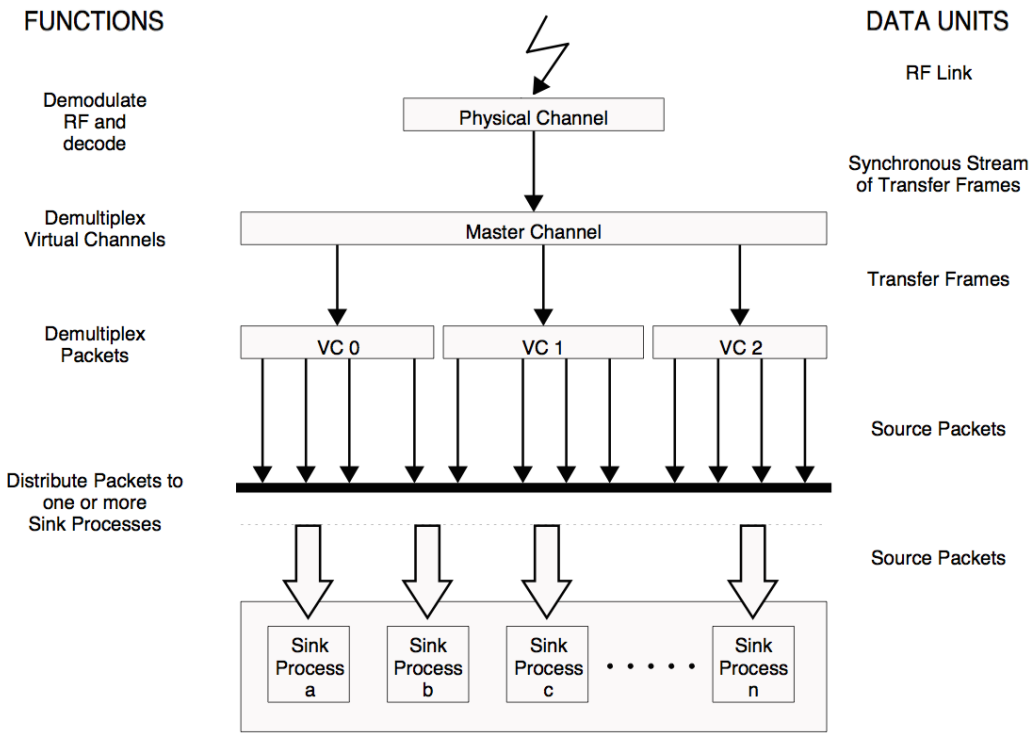


Abb. 4: Datenfluss der Telemetrie vom physikalischen Kanal zur Bodenstation[9]

2.2.2 Softwareebene

Während das zuvor beschriebene CCSDS Protokoll den prinzipiellen Ablauf und das Grundgerüst für die Übertragung definiert, sind – wie bereits zuvor erwähnt – Art und Aufbau der in den CCSDS Paketen enthaltenen Daten völlig beliebig. Da sich jedoch die Anforderungen an die Kommandierbarkeit und Überwachung der Raumfahrzeuge bei den meisten Missionen sehr ähneln, wurde von der ESA der sogenannte Packet Utilization Standard (PUS) definiert, in dem die Kommunikation zwischen Raumfahrzeug und Bodenstation auf Applikationsebene beschrieben ist. Der Standard legt also gewissermaßen eine Software-Schnittstelle fest und definiert als Nebeneffekt den Inhalt der dazu gehörenden Telemetrie- und Telekommando-Pakete.

[9]CCSDS: Packet Telemetry, [ccs95] [S. 2-4, Figure 2.2]

Service Type	Service Name
1	Telecommand verification service
2	Device command distribution service
3	Housekeeping & diagnostic data reporting service
4	Parameter statistics reporting service
5	Event reporting service
6	Memory management service
7	Not used
8	Function management service
9	Time management service
10	Not used
11	Onboard operations scheduling service
12	Onboard monitoring service
13	Large data transfer service
14	Packet forwarding control service
15	Onboard storage and retrieval service
16	Not used
17	Test service
18	Onboard operations procedure service
19	Event-action service

Tabelle 1: PUS Services[11]

Der Standard verfolgt dabei einen servicebasierten Ansatz: Das Raumfahrzeug stellt eine Anzahl von Services zur Verfügung, die wiederum in Subservices unterteilt sind. Jeder Subservice stellt dabei immer entweder eine Input- oder eine Output-Schnittstelle dar, ist also Quelle oder Senke eines ganz bestimmten Telemetrie- bzw. Telekommando-Pakets. Die Services und Subservices sind dabei zur Unterscheidung mit Nummern versehen, wobei die Kombination aus Service Nummer und Subservice Nummern eindeutig ist. Die Nummern 0 bis 127 sind jeweils für standardisierte PUS Services/Subservices reserviert, im Bereich von 128 bis 255 können eigene, missionsspezifische Services und Subservices definiert werden. Zurzeit sind allerdings nur 16 Standard-Services definiert, die Anzahl an Subservices ist bei jedem Service unterschiedlich. Eine Auflistung aller Standard Services findet sich in Tabelle 1. Eine genauere Beschreibung der Services kann dem ECSS Standard E-70-41A[10] entnommen werden. Neben der Möglichkeit eigene Services und Subservices hinzuzufügen, ist es im PUS Standard auch vorgesehen nicht benötigte Services/Subservices wegzulassen. Die PUS Services lassen sich also für jede Mission beliebig anpassen, so dass sie sowohl für kleine als auch große Missionen geeignet sind.

Wie die Onboard Software diese PUS Schnittstelle umsetzt ist dem Entwickler überlassen. In dem zuvor bereits zitierten Buch „Onboard Computers, Onboard Software and

[10] siehe [ecs03]
[11] ECSS-E-70-41A, [ecs03] [S. 50, Table 2]

Satellite Operations: An Introduction“[12] schlägt der Autor Jens Eickhoff auch hierfür einen möglichen Aufbau vor: Die Onboard Software soll für jeden implementierten Service einen entsprechenden Handler bereit stellen, der die Anfragen bearbeitet und die Rückmeldungen generiert (siehe Abbildung 5). Darüber hinaus sind laut Eickhoff ein Event Manager, ein Onboard Control Procedures (OBCP) Manager, ein Onboard Memory Manager, ein Scheduler für die Ausführung zeitgesteuerter Kommandos sowie ein konfigurierbarer Parameter Monitor erforderlich, um die verschiedenen Services zu koordinieren und untereinander zu verbinden.[13] Ein systemübergreifender Kernel sichert, neben zahlreichen anderen Aufgaben, die Kommunikation und Weiterleitung der Daten unter den Service Handlern und anderen Softwarekomponenten.

Interessant für diese Arbeit ist der Standard zum einen, weil die vom DLR entwickelte OBSW und somit auch das Filter-Konzept PUS konform sein soll und zum anderen weil dieser Standard bereits – wie schon in der Einleitung erwähnt – Möglichkeiten zum Filtern der Housekeeping Telemetrie vorsieht. Im Service 3: „Housekeeping and Diagnostic Data Reporting Service“ werden dazu optionale Subservices beschrieben, die diese Funktionalitäten bereitstellen sollen. In Abschnitt 2.3.1 werden diese Subservices genauer erläutert und analysiert.

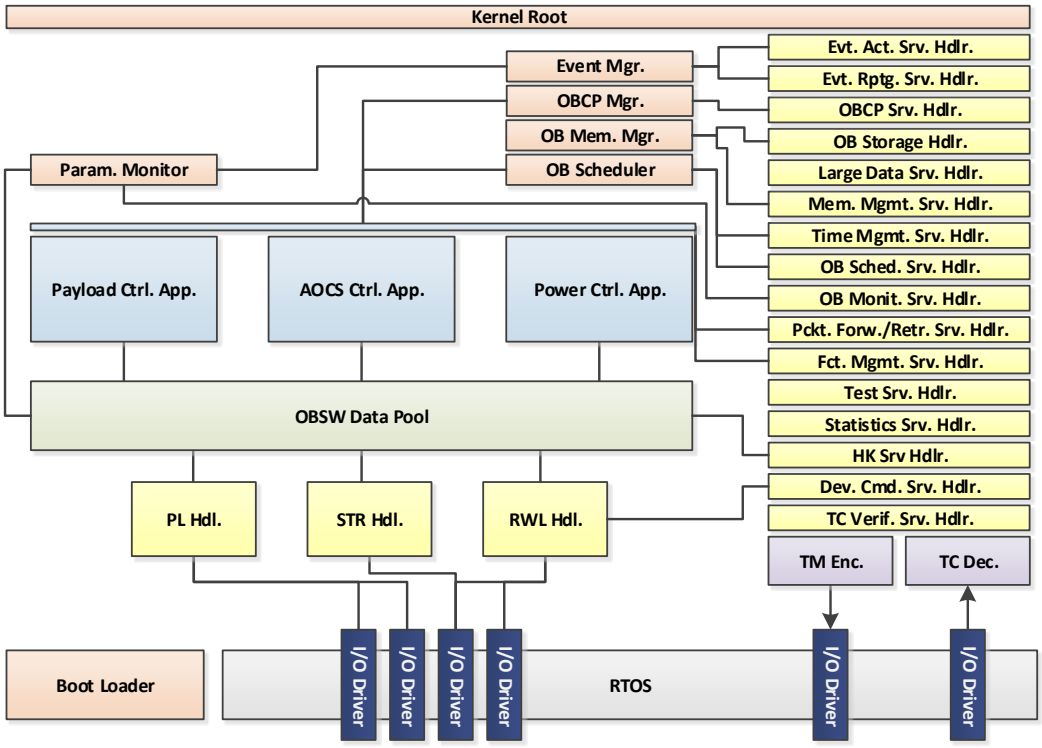


Abb. 5: Onboard Software Service Prinzip[14]

[12] siehe [Eic11]
[13] Eickhoff, vgl. [Eic11] [S. 111]

2.3 PUS Service 3: Housekeeping and Diagnostic Data Reporting Service

2.3.1 Überblick

Der PUS Service 3 „Housekeeping and Diagnostic Data Reporting Service“ dient, wie der Name bereits vermuten lässt, der Generierung von Housekeeping und Diagnostic Reports. Zusammen mit dem „Parameter Statistics Reporting Service“ und dem „Event Reporting Service“ bildet er die Grundlage für die Übermittlung aller zum Betrieb des Raumfahrzeugs nötigen Telemetriedaten. Den Kern des Services bilden sogenannte „Report Definitionen“, in denen festgelegt werden, welche Parameter in welchen Intervallen abgefragt und gesendet werden sollen. Diese Report Definitionen können über entsprechende Subservices aktiviert und deaktiviert sowie – unter der Voraussetzung, dass das Raumfahrzeug diese Funktionen unterstützt – neu erstellt oder gelöscht werden. Darüber hinaus bietet der Service die zuvor mehrfach erwähnten Möglichkeiten einzelne oder mehrere Report Definitionen mit einem Filter zu versehen, so dass diese nicht mehr periodisch sondern nur unter, im Filter festgelegten Bedingungen generiert und gesendet werden. Eine vollständige Auflistung aller Subservices findet sich in Tabelle 2, wobei zusammengehörende Anfragen und Rückmeldungen in der selben Reihe angeordnet sind. Die grau unterlegten Subservices stellen die Minimalanforderungen, die der Service erfüllen muss, dar.

Wie der Tabelle 2 entnommen werden kann, wird in diesem Service zwischen „Housekeeping Reports“ und „Diagnostic Reports“ unterschieden. Housekeeping Reports sind für die reguläre Überwachung des Raumfahrzeugs gedacht, während Diagnostic Reports im Fehlerfall oder bei kritischen Missionsphasen aktiviert werden können. Technisch unterscheiden sich die zwei Report-Arten kaum: Der einzige im Standard festgelegte Unterschied ist, dass Diagnostic Reports mit einer anderen, im Allgemeinen höheren Rate abgefragt und verarbeitet werden. Außerdem können Diagnostic Reports aufgrund der unterschiedlichen Subservice IDs gesondert geroutet werden, so dass sie beispielsweise mit höherer Priorität gesendet werden. Ansonsten sind diese zwei Report-Arten zwar identisch, jedoch streng voneinander getrennt, d.h. es existieren sowohl für Diagnostic als auch Housekeeping Reports eigene Report Definitionen. Die Software, die die Diagnostic und Housekeeping Report Subservices implementiert ist daher auch weitestgehend identisch. Für den Entwurf des Filter-Konzepts ist es aufgrund dieser Äquivalenz der Subservices irrelevant, ob Diagnostic und/oder Housekeeping Reports verwendet werden, weshalb diese Begriffe im Folgenden zum Teil auch synonym verwendet werden.

[14] Zeichnung nach Eickhoff, [Eic11] [S.111, Figure 8.13]

ST	Service requests	ST	Service reports
1	Define New Housekeeping Parameter Report		
2	Define New Diagnostic Parameter Report		
3	Clear Housekeeping Parameter Report Definitions		
4	Clear Diagnostic Parameter Report Definitions		
5	Enable Housekeeping Parameter Report Generation		
6	Disable Housekeeping Parameter Report Generation		
7	Enable Diagnostic Parameter Report Generation		
8	Disable Diagnostic Parameter Report Generation		
9	Report Housekeeping Parameter Report Definitions	10	Housekeeping Parameter Report Definitions Report
11	Report Diagnostic Parameter Report Definitions	12	Diagnostic Parameter Report Definitions Report
13	Report Housekeeping Parameter Sampling-Time Offsets	15	Housekeeping Parameter Sampling-Time Offsets Report
14	Report Diagnostic Parameter Sampling-Time Offsets	16	Diagnostic Parameter Sampling-Time Offsets Report
17	Select Periodic Housekeeping Parameter Report Generation Mode		
18	Select Periodic Diagnostic Parameter Report Generation Mode		
19	Select Filtered Housekeeping Parameter Report Generation Mode		
20	Select Filtered Diagnostic Parameter Report Generation Mode		
21	Report Unfiltered Housekeeping Parameters	23	Unfiltered Housekeeping Parameters Report
22	Report Unfiltered Diagnostic Parameters	24	Unfiltered Diagnostic Parameters Report
		25	Housekeeping Parameter Report
		26	Diagnostic Parameter Report

Tabelle 2: Subservices des Housekeeping and Diagnostic Data Reporting Service[15]

2.3.2 Aufbau von Report Definitionen

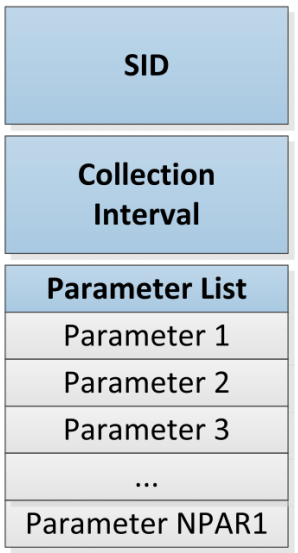


Abb. 6: Struktur einer Report Definition

Eine Report Definition besteht aus einer eindeutigen Identifizierungsnummer (Structure ID, SID), einem Intervall sowie einer beliebigen Anzahl an Parametern, die abgefragt werden sollen (vgl. Abbildung 6). Das Intervall wird dabei nicht in einer Zeiteinheit angegeben, sondern als Vielfaches eines Basisintervalls, der schon beim Bau des Raumfahrzeugs fest vorgegeben wird. Bei einem Basisintervall von 200ms würden beispielsweise alle Parameter einer Report Definition mit dem Intervall 1 im 200 Millisekunden Takt abgefragt und an das Telemetriesystem weiter gereicht werden. Bei einem Intervall von 5 würden die Parameter nur noch im Sekundentakt verarbeitet werden. Somit ist es möglich Reports mit wichtigen Parametern häufig zu generieren, während andere Reports mit weniger wichtigen Parametern nur sporadisch erzeugt werden. Ist das Intervall einer Report Definition abgelaufen, werden alle zu dieser Definition gehörenden Parameter abgefragt und diese Werte zusammen mit der SID und dem Erstellungs-Modus in ein Paket geschrieben. Am Boden kann dann anhand der Service und Subservice IDs erkannt werden, dass es sich um ein Housekeeping bzw. Diagnostic Report Paket handelt und mithilfe der SID und somit der Kenntnis über den Aufbau des Reports die Werte der entsprechenden Parameter zurückgewonnen werden. Der Erstellungs-Modus ist ein optionaler Marker der angibt aus welchem Grund der Report generiert wurde. Unterstützt das Raumfahrzeug lediglich die Generierung periodischer Reports kann dieses Feld weggelassen werden. Besitzt das Raumfahrzeug jedoch auch die Möglichkeit gefilterte Reports zu erstellen, kann über diesen Marker angegeben werden, ob der Report im periodischen Modus, aufgrund eines ausgelösten Triggers oder im gefilterten Modus, jedoch aufgrund eines Timeouts generiert wurde (Erklärung zu Trigger und Timeout siehe nächstes Unterkapitel).

2.3.3 Konzept gefilterter Reports

Filter bestehen ebenfalls aus einer SID mit der angegeben wird, welche Report Definition gefiltert werden soll. Außerdem besitzen sie eine Timeout Zeit und eine beliebige Anzahl an Filter Bedingungen (Trigger). Die Timeout Zeit ist, wie auch schon das Intervall der Report Definitionen, in Vielfachen des Basisintervalls anzugeben und bestimmt die Zeit, nach der ein Report gesendet wird, obwohl keine der Filterbedingung eingetreten ist. Somit kann sichergestellt werden, dass alle Telemetriedaten, auch ohne

[15] ECSS-E-70-41A, [ecs03] [S. 184, Table 43]

Auslösen der Trigger, in regelmäßigen Abständen aktualisiert werden. Typischerweise sind diese Timeout Zeiten allerdings wesentlich größer als die Intervalle des ungefilterten Modus, da das Filtern sonst sinnlos wäre. Die Trigger bestehen wiederum aus jeweils einem Parameter, einem Schwellenwert (Threshold) und einer Angabe über die Art des Schwellenwerts (absolut oder relativ).

Die in den Triggern angegebenen Parameter sollen laut Standard im Basisintervall abgefragt und mit dem Wert der letzten Abfrage verglichen werden. Weicht der Wert eines oder mehrerer Parameter um mehr als den Schwellenwert vom letzten Wert ab, löst der Trigger aus und der dazugehörige Report wird generiert. Wichtig ist zu beachten, dass der ECSS Standard keine Minimum- und Maximumwerte als Filterkriterien vorsieht sondern nur Abweichungen seit der letzten Abfrage. Es kann also beispielsweise angegeben werden, dass ein Report ausgelöst werden soll, wenn sich die Temperatur der CPU seit der letzten Abfrage um mehr als 50% erhöht hat oder wenn sie sich um mehr als 10 °C erhöht hat. Nicht möglich sind dagegen obere und untere Schranken, wie etwa einen Report auslösen, wenn die Temperatur auf über 100 °C gestiegen oder auf unter 0 °C gefallen ist. Solch eine Anweisung könnte über den „Onboard Monitoring Service“ formuliert werden, der jedoch nicht Teil dieser Arbeit ist.

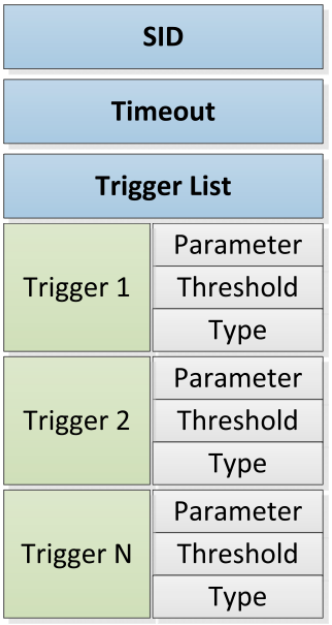


Abb. 7: Struktur eines Filters

2.4 Qualitätsmanagement

2.4.1 C++ Standards und Empfehlungen für eingebettete Systeme

Bei der Entwicklung sicherheitskritischer Software, zu der auch die OBSW von Raumfahrzeugen gezählt wird, müssen einige Einschränkungen und Richtlinien berücksichtigt werden. Es gibt dabei verschiedene Arten von Einschränkungen: Zum einen existieren Sprachelemente und Funktionen deren Einsatz auf eingebetteten Systemen nicht möglich sind, weil die entsprechenden Hardwareressourcen oder die Unterstützung durch das Betriebssystem fehlen. Zum anderen gibt es Sprachelemente, die zwar theoretisch einsetzbar wären, jedoch zu undeterministischen Verhalten führen, weshalb sie aus Sicherheitsgründen nicht oder nur in ganz bestimmten Sonderfällen eingesetzt werden sollten. Außerdem existieren Richtlinien bezüglich Programmier- und Kommentierstil, die zwar in der Regel keine Auswirkungen auf die Architektur und Funktionsweise des Programms haben, jedoch zu besserer Lesbarkeit und der Vermeidung von Fehlern führen sollen.

Die europäische Raumfahrtbehörde ESA hat mit dem „C and C++ Coding Standard“[16] ein umfangreiches Dokument mit solchen Richtlinien und Empfehlungen herausgegeben, von denen einige allgemeingültig, andere dagegen speziell für eingebettete Systeme gedacht sind. Auch die NASA hat diverse Dokumente zur C und C++ Programmierung veröffentlicht, unter anderem den „C++ Coding Standard: Flight Software Branch - Code 582“[17] des Goddard Space Flight Center, der sich explizit auf OBSW bezieht. Aus dem Automobilbereich existiert zudem der sogenannte „Misra C++“[18] Standard, in dem ebenfalls diverse Richtlinien zur Verwendung von C++ in sicherheitskritischen Systemen aufgeführt und erklärt werden. Zwar wurde dieser Standard ursprünglich für die Automobilindustrie entwickelt, jedoch ist er auch in anderen Bereichen, wie etwa der Raumfahrt, anwendbar. Beim DLR existiert zwar kein einheitlicher Standard, es gibt jedoch diverse instituts- und abteilungsinterne Richtlinien, die zum Großteil auf den oben genannten Empfehlungen und Standards basieren. Aufgrund des Umfangs dieser Empfehlungen wird im Folgenden jedoch nur auf die Richtlinien eingegangen, die die Architektur des Filter-Konzepts beeinflussen.

Die wohl wichtigste Einschränkung, die ausnahmslos in allen Standards genannt wird, ist die Vermeidung von dynamischer (Heap-) Speicherbelegung. Zum einen kann die dynamische Speicherbelegung schnell zu nicht oder nur schwer vorhersehbaren Fehlern aufgrund von Speicherüberläufen führen, zum anderen ist das Laufzeitverhalten bei der Allokation des Speichers in der Regel undeterministisch und somit für sicherheitskritische Systeme ungeeignet. Aus dieser Richtlinie ergibt sich auch, dass einige der beliebten Standardbibliotheken nicht oder nur eingeschränkt verwendbar sind, weil sie intern dynamisch Speicher belegen. Es ist bei der Verwendung von Bibliotheken und fremden Codes also immer darauf zu achten, dass dieser ebenfalls für sicherheitskritische Anwendungen geeignet ist. Weitere Einschränkungen existieren bei der Verwendung von Exceptions und RTTI (Run-Time Type Information). Die strikte Vermeidung von Exceptions und RTTI ist lediglich in dem ESA Standard vorgeschrieben. In den Empfehlungen der NASA und in MISRA C++ sind diese Sprachelemente unter bestimmten Voraussetzungen oder, im Falle von RTTI, sogar ohne Einschränkung erlaubt. Diese Empfehlungen können also als „Weak-Recommendations“ betrachtet werden, die zwar nach Möglichkeit eingehalten, jedoch unter bestimmten Umständen auch gebrochen werden können. Eine weitere Empfehlung der ESA betrifft die Simplizität des Codes. Sie besagt, dass Code so simpel wie möglich gehalten und nur optimiert werden soll, falls dies tatsächlich nötig ist. Es ist also nicht erstrebenswert hochgradig optimierte Software zu erstellen, wenn diese Optimierung zu einer erhöhten Komplexität führt und nicht zwingend nötig wäre. Die erhöhte Komplexität steigert lediglich das Risiko

[16] siehe [esa00]
[17] siehe [nas03]
[18] siehe [mis08]

Fehler zu machen und erschwert zudem die Fehlererkennung und -beseitigung. Beim Entwurf von sicherheitskritischer Onboard Software sollte sich der Entwickler also nicht die Frage stellen, ob man den Entwurf noch verbessern könnte, sondern ob man ihn vereinfachen kann. Häufig entsteht durch die Vereinfachung auch automatisch eine Optimierung des Codes.

2.4.2 Test Driven Development

Die *testgetriebene Entwicklung* (engl. Test-Driven Development, TDD) ist eine Methode der Softwareentwicklung, bei der die Tests für eine Software oder Softwarekomponente vor der eigentlich zu entwickelnden Software geschrieben werden. Sie unterscheidet sich damit grundlegend von der herkömmlichen Entwicklungsmethode, bei der die Tests erst nach der Implementierung der zu testenden Software geschrieben werden.

Typischerweise wird TDD auf Klassen- bzw. Modulebene angewandt, d.h. es wird für jede Klasse oder Modul ein eigener, sogenannter Unit-Test angelegt. In der Regel werden dazu *Test-Frameworks* wie beispielsweise JUnit oder Google Test verwendet, die ein automatisiertes Ausführen und häufiges Wiederholen der Tests ermöglichen.

Das Vorgehen bei der testgetriebenen Entwicklung folgt dabei nahezu immer demselben Schema (siehe Abb. 8). Als erstes werden die Anforderungen an das zu entwickelnde Modul festgelegt. Im Anschluss daran werden diese Anforderungen in einen Test übertragen und das Grundgerüst des zu testenden Moduls implementiert, so dass der Test fehlerfrei kompiliert werden kann. Nun wird die eigentliche Funktionalität des Moduls implementiert, getestet und gegebenenfalls so lange korrigiert, bis der Test keine Fehler mehr meldet. Optional können nun noch beliebig viele Optimierungsschritte folgen, wobei nach jeder Änderung erneut getestet werden muss, ob die Funktionalität des Moduls noch gewährleistet ist.[19]

Die Meinungen zur testgetriebenen Entwicklung gegenüber dem herkömmlichen Ansatz gehen stark auseinander. Der testgetriebenen Entwicklung wird häufig nachgesagt zuverlässigere und qualitativ hochwertige Software zu produzieren. Verfechter der traditionellen Methode führen dagegen an, dass TDD lediglich in der Theorie Vorteile besäße und in der Praxis die Entwicklung unnötig verkompliziere. Tatsächlich gab es lange Zeit keine Studien, die die vermeintlichen Vorteile von TDD be- oder widerlegen konnten und auch die heute verfügbaren Studien liefern keine eindeutigen Ergebnisse.

[19] Madeyski, vgl. [Mad10] [S. 226]

Die in dem Buch „Test-Driven Development: An Empirical Evaluation of Agile Practice“ zusammengetragenen Ergebnisse empirischer Studien in Industrie und wissenschaftlichem Umfeld suggerieren, dass die Auswirkungen testgetriebener Entwicklung stark vom Umfeld und der Erfahrung der Programmierer abhängen. Während bei großen Softwarekonzernen wie IBM, Microsoft, StatoilDydro ASA und Ericsson AB positive Effekte bezüglich Fehlerrate, Projektkosten und der Anzahl bestandener Akzeptanztests zu beobachten waren, ergaben die meisten akademischen Studien keine signifikanten Auswirkungen auf die Anzahl bestandener Akzeptanztests. Auch die Entwicklungsgeschwindigkeit wird den Studien zufolge nicht oder sogar negativ beeinflusst. Dafür scheint jedoch die testgetriebene Entwicklung die Generierung einfacherer Klassen und Methoden sowie eine geringere Abhängigkeit der Klassen untereinander zu fördern.[20] Diese verringerten Abhängigkeiten bedeuten, dass mit der testgetriebenen Entwicklung eine modulare und somit einfach zu wartende und wiederverwendbare Softwarearchitektur erzielt werden kann. Außerdem wird durch das Ableiten der Tests aus den Anforderungen sichergestellt, dass die Tests tatsächlich die Einhaltung der Anforderungen verifizieren und nicht nur die Funktionalität des – eventuell nicht den Anforderungen entsprechenden – Codes.[21]

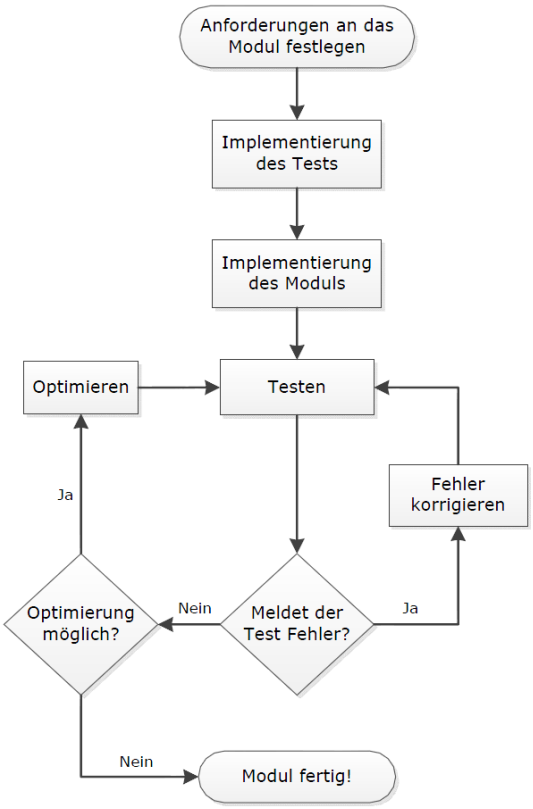


Abb. 8: Entwicklungszyklus der testgetriebenen Softwareentwicklung

Der testgetriebene Entwicklungsansatz bringt also, unter der Voraussetzung, dass er richtig angewandt wird, tatsächlich einige Vorteile mit sich und ist daher insbesondere in der Raumfahrt, in der eine exakte Einhaltung der Anforderungen und ein simples und modulares Softwaredesign den Nachteil der potentiell längere Entwicklungszeit überwiegt, vorzuziehen.

[20] Madeyski, vgl. [Mad10] [S. 15]
[21] Madeyski, vgl. [Mad10] [S. 226]

3 Konzeptentwurf

3.1 Analyse bestehender Software

3.1.1 Tasking Framework

Das *Tasking Framework* ist ein beim DLR entwickeltes Framework, das dem Aufbau eines reaktiven Onboard-Systems auf Basis eines Push-Subscribe-Models dient. Es ist unter verschiedenen Betriebssystemen lauffähig, wobei für die Entwicklung des Filter-Konzepts Linux als Betriebssystem gewählt wurde. Hauptmerkmal des Tasking Frameworks ist das reaktive Scheduling, bei dem Aufgaben, anders als bei den meisten Systemen nicht zeitgesteuert, sondern bei Verfügbarkeit aller benötigten Eingaben ausgeführt werden. Der Vorteil dieses Verfahrens soll am Beispiel einer (vereinfachten) „Sensor-Fusion“ Berechnung, wie sie unter anderem beim AOCS zum Einsatz kommt, veranschaulicht werden: Sensor-Fusion bezeichnet ein Verfahren, bei dem Daten von verschiedenen Sensoren zusammengeführt werden um daraus eine einzige, verlässlichere Information zu generieren. So könnten beispielsweise die Daten von einem Gyroskop, einem Magnetfeldsensor und einer Sternkamera verwendet werden, um damit möglichst präzise die Lage des Raumfahrzeugs bestimmen zu können. Diese Berechnungen sind jedoch häufig sehr komplex und nehmen viel Zeit in Anspruch, besonders wenn zunächst, wie beim herkömmlichen, zeitgesteuerten Scheduling Verfahren, auf die Daten aller Sensoren gewartet werden muss bevor sie zusammengeführt werden können (vgl. Abb. 9, A). Häufig lassen sich solche Berechnungen jedoch in mehrere Schritte unterteilen, die zum Teil bereits ausgeführt werden könnten bevor die Daten der restlichen Sensoren zur Verfügung stehen. Bei Gyroskopen und Magnetfeldsensoren ist es zum Beispiel sinnvoll einen simplen Algorithmus, der die durch Rauschen im Sensor entstandenen Messspitzen heraus filtert, anzuwenden, bevor die Daten weiter verarbeitet werden. Dazu könnte man beispielsweise warten bis 10 Werte des jeweiligen Sensors verfügbar sind, anschließend die größten und kleinsten Werte aus dieser Messreihe streichen und aus den restlichen Werten einen Mittelwert berechnen. Somit kann bereits der durch statistische Streuung der Messwerte entstandene Fehler zu einem großen Teil kompensiert werden. Ist dann auch noch die Lageinformation des Sternsensors verfügbar, können alle Daten zusammengeführt und eine einzige, sehr genaue Aussage über Lage des Raumfahrzeugs getroffen werden.
Wie in Abbildung 9 erkennbar ist, lässt sich so gegenüber einem herkömmlichen Scheduling Verfahren erheblich Rechenzeit sparen, insbesondere wenn die Algorithmen wie im AOCS in einer Schleife oft hintereinander ausgeführt werden. Zwar ließe sich auch bei einem System mit herkömmlichem Scheduler solch eine Sensor-Fusion Berechnung in Teilschritte aufteilen, die Koordination der zu unterschiedlichen Zeitpunkten eintreffenden Daten wäre jedoch wesentlich komplexer und würde vermutlich den Geschwindigkeitsvorteil zunichte machen.

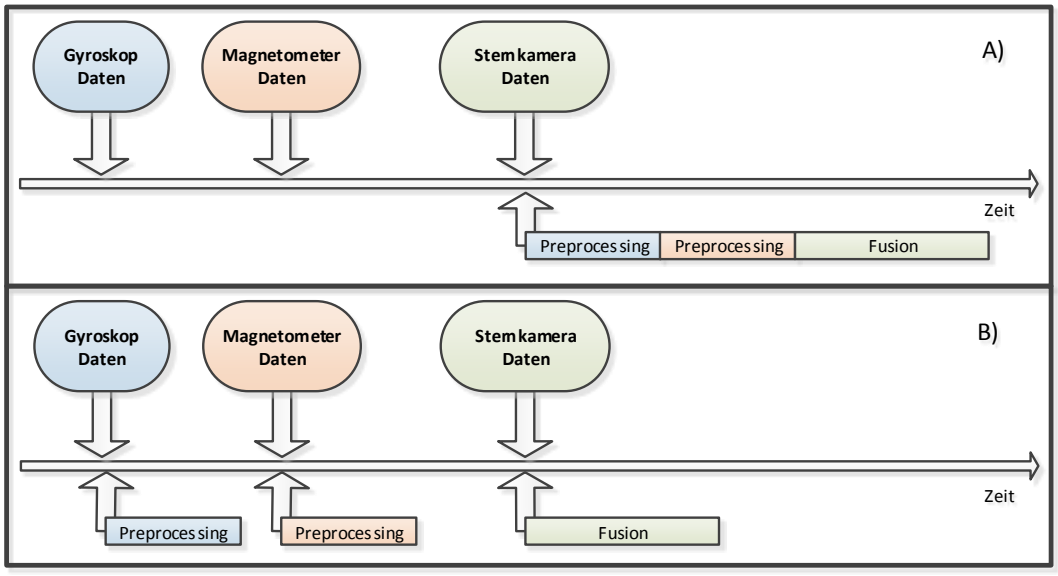


Abb. 9: Gegenüberstellung der Laufzeit von herkömmlichen (A) und reaktivem Scheduling (B) am Beispiel einer Sensor-Fusion Berechnung

Um solch ein reaktives System aufbauen zu können, bietet das Tasking Framework dem Entwickler verschiedene Elemente. Die Wichtigsten davon sind „Task“, „TaskMessage“ und „TaskInput“. Mithilfe von Tasks können die eigentlichen Aufgaben und Algorithmen implementiert werden. Dazu muss lediglich eine von Task abgeleitete Klasse erstellt und ihre „execute“-Methode überschrieben werden. Einem Task können beliebig viele (Task-)Inputs zugeordnet werden, die die Ausführung der execute-Methode auslösen können. Normalerweise wartet ein Task mit der Ausführung bis alle Inputs aktiviert wurden. Alternativ kann ein Input auch so konfiguriert sein, dass er den Task sofort, unabhängig vom Status der anderen Inputs, auslöst. Die Inputs besitzen einen „Activation Threshold“ also einen Zahlenwert, der angibt nach wie vielen neu eingegangenen Nachrichten der Input aktiviert werden soll. Ein Input stellt immer eine 1:1 Verbindung zwischen einem Task und einer TaskMessages dar, das heißt ein Input ist immer exakt einem Task und einer TaskMessage zugeordnet. Umgekehrt können, wie schon erwähnt, Tasks beliebig viele Inputs besitzen und auch eine TaskMessage kann im Falle einer neuen Nachricht mehrere Inputs darüber informieren. TaskMessages besitzen eine „push“-Methode, über die der Input aktiviert werden kann. Wichtig ist zu beachten, dass TaskMessages zwar einen Mechanismus zum Aktivieren von Inputs/Tasks besitzen, jedoch von Haus aus keine Daten enthalten. Um mit den TaskMessages tatsächlich Daten an den Task weiterleiten zu können, muss eine von TaskMessage abgeleitete Klasse erstellt, ein entsprechender Datentyp in Form einer Variablen oder einer Datenstruktur hinzugefügt sowie die push-Methode überschrieben werden. Somit können TaskMessages flexibel für verschiedene Datentypen eingesetzt werden. Darüber hinaus

gibt es im Tasking Framework noch weitere Elemente wie etwa ein „TaskEvent“, mit dem Messages periodisch generiert werden können oder „TaskSets“, mit denen Tasks miteinander verbunden werden können, so dass sie erst reaktiviert werden können nachdem alle Tasks eines TaskSets einmal ausgeführt wurden. Da sie für das Grundverständnis des Frameworks und das Filter Konzept nicht relevant sind werden sie hier jedoch nicht weiter erläutert.

Beim oben genannten Beispiel der Sensor-Fusion wären die von den Sensoren kommenden Rohdaten in TaskMessages, bzw. einer von TaskMessage abgeleiteten Klasse enthalten. Sind neue Daten verfügbar werden die entsprechenden Inputs aktiviert. Bei den Inputs der Gyroskope und Magnetometer wäre es sinnvoll die Inputs so zu konfigurieren, dass sie den dazugehörigen Task erst nach Eintreffen von beispielsweise 10 Nachrichten auslösen. Im Anschluss kann der Task aus diesen 10 Werten die Messspitzen heraus filtern und einen Mittelwert bilden. Sind die so vorverarbeiteten Daten der Gyroskope und Magnetometer sowie die Lageinformation des Sternsensors verfügbar, wird der Task zum Zusammenführen der Daten aktiviert. Dieser Task könnte das Produkt der Sensor-Fusion Berechnung wiederum über Messages an andere Tasks weitergeben, etwa zum Berechnen der nötigen Drehzahl der Reaction Wheels (vgl. Abbildung 10).

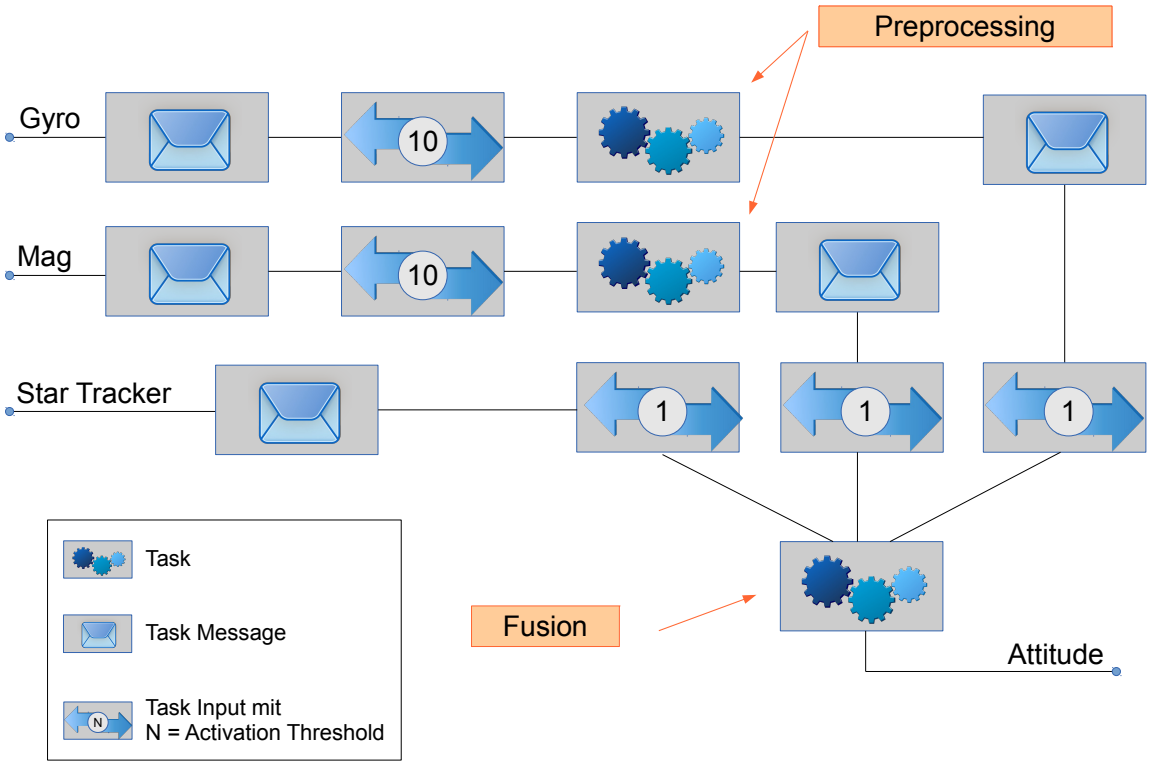


Abb. 10: Veranschaulichung der Arbeitsweise des Tasking Frameworks

3.1.2 Bestehende Housekeeping & Diagnostic Report Implementierung

Um einen Filter entwickeln zu können ist ein Housekeeping/Diagnostic Report System erforderlich, auf das der Filter angewendet werden kann. Leider existiert beim DLR kein eigenständiges, wiederverwendbares Housekeeping/Diagnostic Framework, das für diesen Zweck verwendet werden könnte. Zwar wird bestehende Software bei neuen Missionen zum Teil wiederverwendet, jedoch sind diese Housekeeping und Diagnostic Report Systeme tief in die restliche Onboard Software verwurzelt und können ohne den Rest der OBSW nicht bzw. nur mit einigen Anpassungen kompiliert und getestet werden.

Um trotzdem ein realitätsnahes Housekeeping & Diagnostic Report System zu erhalten auf dessen Grundlage das Filter-Konzept entwickelt und getestet werden kann, wurde ein Teil des Diagnostic Report Systems der aktuellen Eu:CROPIS Mission verwendet und so modifiziert, dass es als eigenständiges Testsystem kompilierbar und ausführbar ist. Die so entstandene Software ist zwar nicht mehr direkt als Onboard Software für reale Systeme verwendbar, der Aufbau ist jedoch sehr ähnlich, weshalb sowohl der Entwurf als auch der Prototyp des Filter-Konzepts problemlos auf echte Onboard Software übertragbar sein sollte.

Der ursprüngliche Aufbau dieser Diagnostic Report Implementierung ist als Klassendiagramm im Anhang A dargestellt. Die „DiagnosticReportService“ Klasse dient als Schnittstelle zum Rest des Onboard Systems. Über diese Klasse wird der „DiagnosticReportManager“ gesteuert, der eine Menge von „DiagnosticReportDefinition“ Instanzen verwaltet. In den DiagnosticReportDefinition Instanzen sind die eigentlichen Informationen über die zu erstellenden Reports gesichert, also welche Parameter in welchem Intervall abgefragt werden sollen. Um keinen dynamischen Speicher verwenden zu müssen, wird bereits beim Entwurf der Software eine maximale Anzahl an Report Definitionen vorgegeben, die in jedem Fall fest im Speicher reserviert sind. Wird versucht mehr als diese maximale Anzahl an Report Definitionen zu erstellen wird die Anfrage abgewiesen. Da eine Report Definition wiederum beliebig viele Parameter (genauer gesagt „DiagnosticParameter“) umfassen kann, wäre an dieser Stelle erneut dynamische Speicherzuweisung nötig. Um dies zu umgehen besitzt jede Report Definition wieder eine fest vorgegebene, maximale Anzahl an DiagnosticParametern, die gespeichert werden können. Wird versucht eine Report Definition zu erstellen, die mehr DiagnosticParameter als diese maximale Anzahl umfasst, wird die Report Definition ebenfalls abgewiesen. Ein DiagnosticParameter besteht im Wesentlichen lediglich aus einer 32 Bit großen ID, die angibt welcher Wert abgefragt werden soll. Dieser Wert kann entweder ein „Parameter“ oder eine „RelayMessage“ sein. An dieser Stelle sei auf die etwas unglückliche Namensgebung „Parameter“ und „DiagnosticParameter“ hin-

gewiesen: Ein DiagnosticParameter verweist lediglich auf einen Parameter oder eine RelayMessage. Parameter und RelayMessages enthalten dann tatsächlich die Daten, die beim Erstellen des Reports abgefragt werden. Ein Parameter ist ein Wert, der sich im regulären Betrieb des Raumfahrzeugs nicht ändert, wie zum Beispiel Konfigurationen und Offset-Werte für Sensoren. Sie sind daher für das Filter-Konzept nicht weiter relevant. RelayMessages dagegen sind eine von TaskMessage abgeleitete Klasse, die um eine eindeutige ID und eine ID-Verwaltung sowie einen Mechanismus zum Serialisieren der enthaltenen Daten ergänzt wurden. Sie enthalten zwar, wie auch schon die TaskMessages, zunächst keine Daten und müssen daher für die Verwendung überladen werden, sie besitzen jedoch ein Objekt („SerializableObject“), über das später die enthaltenen Daten abgefragt und in ein Housekeeping/Diagnostic Paket geschrieben werden können. Somit lassen sich RelayMessages genauso wie TaskMessages zum Benachrichtigen und Transportieren von Daten zu den Tasks einsetzen, jedoch bieten sie darüber hinaus die Möglichkeit ihre Daten für Reports zur Verfügung zu stellen. Die letzte, wichtige Klasse des Diagnostic Report Systems stellt der „DiagnosticReportTask“ dar. Wie der Name vermuten lässt ist diese Klasse von der Task Klasse des Tasking Frameworks abgeleitet und stellt somit eine, durch eine TaskMessage auslösbare, Funktion dar. In diesem Fall wird der Task durch ein TaskEvent periodisch im Basistakt (siehe 2.3.2) aufgerufen und überprüft bei allen aktiven Report Definitionen, ob ihr Intervall abgelaufen ist und ein entsprechender Report generiert werden soll. Ist dies der Fall werden alle in der Report Definition aufgeführten Parameter und RelayMessages abgefragt, serialisiert und in einen, vom Telemetrie Interface bereitgestellten, Speicherbereich geschrieben.

Um das Diagnostic Report System auch ohne den Rest der OBSW kompilieren und testen zu können waren, wie zuvor schon erwähnt, einige Anpassungen notwendig. In erster Linie musste der Serialisierer und das Telemetrie Interface ersetzt werden, die für Datenverwaltung und -austausch innerhalb der OBSW notwendig sind. Als (De-)Serialisierer wurde eine einfache Klasse implementiert, die beliebige primitive Datenstrukturen an einen festgelegten Speicherbereich schreiben und später wieder zurück in einen strukturierten Datentyp wandeln kann. Der Serialisierer kann ähnlich wie der original Serialisierer der OBSW verwendet werden, erhebt jedoch, da er lediglich zum Testen geschrieben wurde, keinerlei Anspruch auf Fehlerfreiheit und Absturzsicherheit. Das Telemetrie Interface wurde ebenfalls durch ein simples „Dummy-Interface“ ersetzt, das zum Schreiben der Reports ein Datenfeld fester Größe zur Verfügung stellt und dieses, nach Erstellung des Reports, über den Standardoutput an die Konsole ausgibt. Eine weitere Anpassung betrifft die DiagnosticReportService Klasse. Sie stellt, wie bereits zuvor erwähnt, die Schnittstelle des Diagnostic Report Systems zum Rest der OBSW dar und wurde daher restlos weggelassen. Für die Demo Anwendungen kann auch direkt der DiagnosticReportManager angesprochen werden.

3.2 Anforderungen

Die Anforderungen an das zu entwickelnde Konzept, beziehungsweise den Prototypen lassen sich direkt aus der Aufgabenstellung und den in den Grundlagen aufgeführten Einschränkungen und Richtlinien für Onboard Computer und Onboard Software ableiten. Eine Auflistung dieser Anforderungen findet sich in Tabelle 3.

Nummer	Anforderung	Begründung
1	Der Filter soll in C++ geschrieben werden und auf dem Tasking-Framework aufsetzen.	Siehe Aufgabenstellung.
2	Der Filter soll dem im ECSS Standard E-70-41A beschriebenen Konzept entsprechen. Eine vollständige Umsetzung des Standards ist jedoch nicht gefordert.	Siehe Aufgabenstellung.
3	Für das Filterkonzept sollen, soweit möglich und sinnvoll, die in Abschnitt 2.4.1 aufgeführten Empfehlungen für die Entwicklung sicherheitskritischer Echtzeitsoftware in C++ eingehalten werden. Insbesondere soll keine dynamische Zuweisung von Heap Speicher stattfinden.	Siehe Abschnitt 2.4.1
4	Die Entwicklung soll dem „Test Driven Development“ Ansatz verfolgen.	Siehe Abschnitt 2.4.2
5	Der Filter soll so wenig Ressourcen wie möglich verbrauchen (Speicher- und Laufzeitbedarf), ohne dabei die Übersichtlichtlichkeit und Simplizität zu stark zu beeinträchtigen.	Siehe Abschnitt 2.1.1 und 2.4.1

Tabelle 3: Anforderungen an das Filter-Konzept und den Prototypen

Der Begriff „Anforderungen“ mag an dieser Stelle eventuell irreführend sein, da die Anforderungen, anders als sonst in der Raumfahrtbranche üblich, eher vage gehalten und zum Teil nicht eindeutig verifizierbar sind. Dies hat den Grund, dass es sich bei der zu erstellenden Software um ein Konzept und nicht etwa um einen Teil einer konkreten Mission handelt, wodurch es schwierig bzw. nicht sinnvoll ist feste Anforderungen zu

stellen. Da die in Tabelle 3 aufgeführten Punkte jedoch Bedingungen an das Konzept stellen, die maßgeblich dessen Architektur beeinflussen wurde der Begriff „Anforderungen“ trotzdem als passend erachtet.

3.3 Konzept/Modellierung

3.3.1 Überblick

Das vollständige Klassendiagramm des Filter-Konzeptentwurfs kann Anhang B entnommen werden. Das Konzept sieht vor, eine von RelayMessage abgeleitete Klasse „ExtendedRelayMessage“ zu erstellen, die zusätzlich zu den von RelayMessage geerbten Eigenschaften eine Schnittstelle zum einheitlichen Prüfen einer Filterbedingung bietet. Von dieser ExtendedRelayMessage wird wiederum ein Klassen-Template „FilterableMessage“ abgeleitet, das es nach wie vor erlaubt Message Klassen mit beliebigen Datencontainern zu erstellen und trotzdem die Möglichkeit bietet bereits einige Funktionalitäten, ohne genaue Kenntnis des verwendeten Datentyps, zu implementieren. Darüber hinaus müssen die Filter Definitionen in irgendeiner Form gespeichert und verwaltet werden. Zum Speichern der Filterdefinitionen wurde eine eigene Klasse („FilterDefinition“) erstellt, die sich am Aufbau der ReportDefinition Klasse orientiert. Zum Verwalten der Definitionen wurde der bereits existierende ReportDefinitionManager erweitert. Auch die bereits vorhandene DiagnosticReportTask Klasse muss modifiziert werden, so dass sie nicht nur das periodische Generieren und Senden von Reports, sondern auch die gefilterte Generierung unterstützt. In den folgenden Unterkapiteln werden die einzelnen Elemente des Filterkonzepts detaillierter beschrieben, die entsprechenden Design Entscheidungen begründet sowie mögliche Alternativen aufgezeigt.

3.3.2 Filterbare Messages

Der Entscheidung den Funktionsumfang der RelayMessages durch die ExtendedRelayMessage Klasse und dem FilterableMessage Klassen-Template zu erweitern gingen eine Reihe von Problemstellungen voraus:

Wie in Abschnitt 2.3.3 bereits erwähnt wurde, sieht der im Packet Utilization Standard beschriebene Subservice zum Filtern diagnostischer Reports als Filterkriterium keine absoluten Grenzen, sondern die relative Abweichungen eines Parameters zwischen zwei Abfragen vor. Es sind also zumindest die letzten zwei Werte derselben Message nötig, um die relative Abweichung bestimmen zu können. Da dieselbe Message jedoch von verschiedenen Report Definitionen zu unterschiedlichen Zeitpunkten abgefragt werden könnte, müsste man für jede Message und jeden möglichen Abfrageintervall Speicher reservieren, um eine dynamische Speicherallokation zu vermeiden. Dies ist zum einen unpraktisch, zum anderen bietet es keinen nennenswerten Mehrwert, die früheren Werte

einer Message zu mehr als einem Zeitpunkt zu speichern. Daher wurde für das Filter-Konzept eine etwas andere Vorgehensweise gewählt: Der aktuelle Wert wird, anders als im Standard beschrieben, nicht mit dem Wert der vorherigen Abfrage verglichen, sondern mit dem zuvor über die push-Funktion der Message übermittelten Wert. Da die Onboard Software von Raumfahrzeugen sowieso meist zyklisch mit einem gewissen Basistakt abläuft, ist die Abweichung dieser Vorgehensweise zum PUS Standard relativ gering. Zeitliche Abfragen, wie etwa „Abweichung pro Sekunde“ lassen sich unter der Voraussetzung, dass der Intervall der push-Aufrufe bekannt ist, nach wie vor formulieren. Außerdem lassen sich auf diese Art und Weise die Vorteile des Tasking Frameworks besser ausnutzen. Wie in dem Vergleich zwischen dem traditionellen Scheduling Verfahren und dem des Tasking Frameworks deutlich wurde, liegt die Stärke des Tasking Frameworks gerade darin, nicht zeitgesteuert auf alle Eingaben warten zu müssen, sondern Berechnungen auszuführen sobald alle benötigten Eingaben vorhanden sind. Dementsprechend wäre es in dieser Umgebung unter Umständen auch sinnvoller, statt Filterbedingungen wie „Abweichung pro Sekunde“ besser „Abweichung pro Ausführung“ zu wählen, um die zeitliche Abhängigkeit zu eliminieren. Dies lässt sich durch den Vergleich der beiden zuletzt übermittelten Werte viel besser realisieren, als durch das im Standard vorgesehene zeitgebundene Vergleichen. Um den aktuellen Wert mit dem zuletzt übermittelten Wert vergleichen zu können müssen die Message-Klassen daher die Möglichkeit besitzen, die beiden zuletzt übergebenen Werte speichern zu können. Gleichzeitig soll die Fähigkeit der RelayMessages gewahrt bleiben, beliebige Datentypen und -strukturen transportieren zu können. Auch hierfür gäbe es prinzipiell zwei Lösungsmöglichkeiten: Man könnte, wie auch schon bei den RelayMessages, überhaupt keine Daten in die FilterableMessage integrieren und es in die Verantwortung des Programmierers neuer Message-Arten legen, anstatt nur einer Variable gleich zwei Variablen des gewünschten Datentyps zu erstellen, so dass sowohl der alte als auch der neue Wert der Message gespeichert werden können. Allerdings wäre es in diesem Fall auch nicht möglich, der Klasse irgend eine Funktionalität zu verleihen. Alle Funktionen wären rein virtuell und müssten für jede abgeleitete Message-Art neu implementiert werden. Dabei gäbe es Funktionen, die generisch implementiert werden könnten, wenn die entsprechenden Variablen bereits deklariert wären. Die push-Funktion muss beispielsweise immer – unabhängig des Datentyps – den neu erhaltenen Wert abspeichern, während der ehemals aktuelle Wert an die Stelle des alten Werts geschrieben wird. Außerdem muss sichergestellt werden, dass der aktuelle und der alte Wert einer Message nur dann verglichen werden, wenn tatsächlich auch bereits ein aktueller und ein alter Wert vorhanden sind (mindestens 2 Aufrufe von push nötig). Diese Funktionen jedes mal erneut zu implementieren stellt sowohl einen unnötigen Aufwand als auch eine potentielle Fehlerquelle dar, weshalb für das Filter Konzept die zweite Möglichkeit das Problem zu lösen gewählt wurde: Die Verwendung eines Klassen-Templates. Mithilfe eines Klassen-Templates kann eine Klasse mit einem noch nicht spezifizierten

Datentyp T implementiert werden. Innerhalb dieser Klasse ist es dann möglich T wie einen ganz gewöhnlichen Datentyp zu verwenden, man kann also Variablen vom Typ T erstellen oder T als Datentyp für ein Argument oder Rückgabewert verwenden. Wichtig ist zu beachten, dass ein Klassen-Template zunächst keine Klasse darstellt, erst durch die Instanziierung des Templates durch die Angabe eines konkreten Datentyps erzeugt der Compiler eine reale Klasse. Außerdem muss darauf geachtet werden, dass alle Datentypen, mit denen das Template instanziiert wird auch die, in dem Klassen-Template verwendeten Operatoren unterstützt. Das FilterableMessage Template darf beispielsweise nur mit Datentypen verwendet werden, die den Zuweisungsoperator (=) unterstützen, da dieser innerhalb der push-Funktion benötigt wird. Da jedoch die meisten Datentypen den Zuweisungsoperator unterstützen und der Compiler im Falle eines nicht kompatiblen Datentyps bereits beim Kompilieren eine Fehlermeldung generiert, ist diese Einschränkung zu vertreten.
Aus der Verwendung eines Klassen-Templates ergibt sich jedoch noch ein anderes Problem. Anders als die zuvor genannten Funktionen, die generisch für alle FilterableMessages implementiert werden können, ist es nicht möglich für alle Datentypen eine einheitliche Überprüfung von Trigger Bedingungen zu erstellen. Es ist also erforderlich, für jeden Datentypen (und somit für jede Template Instanz) eine eigene Funktion zum Überprüfen der Filter Bedingung zu implementieren. Die ließe sich leicht mithilfe einer rein virtuellen „trigger“-Funktion lösen, so dass für jede neue Message Art eine neue Klasse erstellt werden muss, die von der entsprechend instanziierten Template-Klasse erbt und die trigger-Funktion mit ihrer eigenen, datentypspezifischen Implementierung überschreibt. Allerdings sind alle von einem Template instanziierten Klassen eigenständige Klassen ohne gemeinsame Basis, es wäre also nicht möglich die trigger-Funktion der verschiedenen Message Arten über eine gemeinsame Schnittstelle anzusprechen, wenn die trigger-Funktion in dem Template deklariert würde. Aus diesem Grund wurde die ExtendedRelayMessage erstellt, die keinen anderen Zweck verfolgt, als eine rein virtuelle „isTriggered“-Funktion zu deklarieren, so dass die isTriggered-Funktion der verschiedenen FilterableMessage Instanzen von einer gemeinsamen Basis Klasse aus aufgerufen werden können, ohne Kenntnis über die Art der FilterableMessage zu haben. Der Grund, warum hier einmal von einer trigger-Funktion und einmal von einer isTriggered-Funktion gesprochen wird ist, dass die FilterableMessage Klasse tatsächlich beide Funktionen besitzt. Die isTriggered-Funktion überschreibt die isTriggered-Funktion der ExtendedRelayMessage und bietet somit die gemeinsame Schnittstelle. Die trigger-Funktion ist jene rein virtuelle Funktion, die überschrieben werden soll, um die Entscheidung, ob eine Filter Bedingung zutrifft oder nicht, für einen bestimmten Datentyp zu definieren. Die Verwendung zweier Funktionen ist nötig, um die Überprüfung, ob die zu vergleichenden Werte überhaupt definiert sind, implementieren zu können. In der isTriggered-Funktion der FilterableMessage wird zunächst geprüft, ob diese Bedingung erfüllt ist, bevor die trigger-Funktion aufgerufen wird. Ist dies nicht

der Fall wird, ohne die trigger-Funktion aufgerufen zu haben, „false“ zurück gegeben, da nicht genügend Werte gleichzeitig bedeuten, dass es noch nicht genügend (gemessene) Änderungen gegeben hat, die die Auslösung eines Reports rechtfertigen würden.

Im vorangegangen Abschnitt wurde behauptet, dass für jeden Datentyp eine eigene trigger-Funktion vonnöten ist, ohne dies zu begründen. Außerdem könnte man sich die Frage stellen, warum die Überprüfung, ob eine Filterbedingung eingetreten ist überhaupt in der FilterableMessage Klasse stattfinden sollte und nicht etwa in einer eigenen Klasse, der lediglich ein Pointer auf die zu überprüfende Message übergeben wird. Um dies zu beantworten, muss zunächst die grundlegende Frage betrachtet werden, wie der Unterschied zwischen zwei Werten eines beliebigen Datentyps definiert ist und damit zusammenhängend, von welchem Datentyp der Schwellenwert für diesen Unterschied sein soll. Bei Zahlenwerten ist diese Frage einfach zu beantworten: Um den absoluten Unterschied zu erhalten, müssen die zwei Werte lediglich subtrahiert werden und auch das Umrechnen in einen prozentualen Unterschied gestaltet sich einfach. Bei allen anderen Datentypen ist diese Frage dagegen nur schwer bzw. gar nicht zu beantworten. Ohne zusätzliche Angaben lässt sich beispielsweise nicht sagen, ob sich ein Vektor um mehr als 50% verändert hat. Man könnte lediglich überprüfen, ob sich zum Beispiel die Länge eines Vektors oder der Winkel den der Vektor zu einer bestimmten Ebene hat um mehr als 50% verändert hat. Diese Werte (Länge, Winkel) sind allerdings wieder Zahlenwerte. Ähnliche Beispiele lassen sich auch für andere Datentypen finden. Es bleibt also festzuhalten, dass lediglich die Angabe von Zahlenwerten sinnvoll für einen Schwellenwert sind, weshalb im Filterkonzept alle Schwellenwerte vom Typ double sind. Der eigentliche Sinn der trigger-Funktion liegt also darin, den in dieser Message enthaltenen Datentyp auf einen Zahlenwert zu reduzieren, bevor er gegen einen Schwellenwert getestet wird. Daher ist es auch sinnvoll, die trigger-Funktion in die Message zu integrieren, da somit alle zu einem Datentyp bzw. einer Message gehörenden Funktionen in einer Klasse enthalten sind und keine unerwünschten Abhängigkeiten entstehen. Der Schwellenwert selbst sowie die Angabe, ob der Schwellenwert als absoluter oder prozentualer Wert zu interpretieren ist, werden der trigger-Funktion als Argument übergeben, so dass dieselbe Message von verschiedenen Filtern auf verschiedene Schwellenwerte hin geprüft werden kann. Ein kurzes Beispiel, das zeigt wie FilterableMessages konkret verwendet werden, findet sich in Kapitel 4.

Das zuvor beschriebene FilterableMessage Klassen-Template würde bereits alle nötigen Funktionen liefern, um ein wirksames Filter System umzusetzen. Betrachtet man allerdings das Sensor-Fusion Beispiel aus Abschnitt 3.1.1, zeigt sich eine potentielle Schwäche dieser Implementierung. In dem Beispiel wurde eine (vereinfachte) Sensor-Fusion Berechnung beschrieben, bei der ein TaskInput so konfiguriert wurde, dass er 10 Aufrufe der push-Funktion abwartet bis er den dazugehörigen Task ausführt. Dies ist

selbstverständlich nur sinnvoll, wenn in der Message die letzten 10 Werte gespeichert werden können, beispielsweise in Form einer FIFO (First In, First Out) Datenstruktur fester Länge, die automatisch das älteste Element löscht, wenn sie voll ist und ein neues Element eintrifft. Würde man allerdings solch eine FIFO Klasse erstellen, um damit eine FilterableMessage zu instanziieren, würde die entstandene FilterableMessage Klasse zwei FIFO Objekte enthalten, die bis auf das erste und letzte Element einen identischen Inhalt besitzen, das zweite FIFO Element wäre somit verschwendeter Speicher. Außerdem gestaltet sich das Hinzufügen neuer Elemente zu dem FIFO Speicher als umständlich. Man müsste zuerst das Element in ein lokal verfügbares FIFO Objekt pushen, bevor anschließend das gesamte FIFO Objekt in die FilterableMessage gepusht wird. Dies ist nicht zielführend und angesichts der Tatsache, dass FIFO Objekte in Kombination mit dem Tasking Framework häufig verwendet werden, nicht akzeptabel. Allerdings bietet sich auch eine relativ einfache Lösung für dieses Problem: Betrachtet man die ursprüngliche FilterableMessage genauer, fällt auf, dass die bisherige Implementierung bereits eine Art FIFO Struktur mit einer festen Größe von zwei bietet. Schließlich stellt das Speichern der letzten zwei Werte nichts anderes als eine FIFO Struktur dar. Erweitert man das Klassen-Template so, dass nicht nur ein Datentyp, sondern auch die Anzahl der gespeicherten Elemente angegeben werden kann, stellt das FilterableMessage Klassen-Template automatisch einen FIFO Speicher beliebiger Größe dar. Wird keine Größe angegeben geht der Compiler automatisch von einer Standardgröße von zwei aus, so dass diese Zusatzfunktion nicht weiter beachtet werden muss, solange sie nicht gebraucht wird. Die Möglichkeit die Anzahl der gespeicherten Elemente manuell zu bestimmen erlaubt es theoretisch auch andere Datenstrukturen, wie etwa Stacks in einer FilterableMessage zu implementieren, allerdings müsste in diesem Fall auch die push-Funktion überschrieben werden. Da solche Datenstrukturen jedoch selten bis nie in Messages vorkommen, ist es durchaus sinnvoll die push-Funktion für einfache Datentypen und FIFOs vorzubereiten, während sie für andere Datenstrukturen die Option des Überschreibens bleibt.

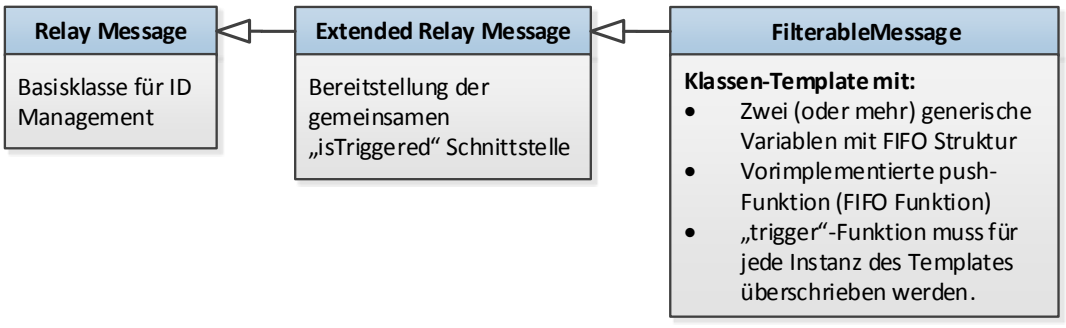


Abb. 11: Struktur filterbarer Nachrichten (FilterableMessages)

3.3.3 Filter Definitionen, Filterbedingungen und erweiterter Diagnostic Report Manager

Die Filter Definitionen sind im Aufbau den Report Definitionen nachempfunden. Ähnlich wie die Report Definitionen, die eine Menge an leeren DiagnosticParametern besitzen, die beim Einlesen der Report Definition gefüllt werden, besitzen die Filter Definitionen eine feste Anzahl an ThresholdDefinitions, die die Filterbedingung, also Parameter ID, Schwellenwert und Schwellenwerttyp, enthalten. Somit kann eine Filter Definition zwischen einer und einer zur Kompilierzeit festgelegten Anzahl maximaler Filterbedingungen (Trigger) enthalten. Da es nur Sinn macht maximal einen Filter pro Report zu erstellen, ist die maximale Anzahl an Filter Definitionen automatisch durch die maximale Anzahl an Report Definitionen gegeben. Umgekehrt stellt sich die Frage, ob man nicht auch mit weniger Filtern auskommen kann, d.h. ob es eventuell Report Definitionen gibt, die nie gefiltert sondern nur periodisch erstellt werden sollen. In dem Prototyp des Konzepts wurde aus Gründen der Übersichtlichkeit eine 1:1 Verbindung realisiert, was bedeutet, dass jede Report Definition exakt eine Filterdefinition besitzt. Sollte es bei einer Mission als sinnvoll erachtet werden, Report Definitionen ohne Filtermöglichkeit bereit zu stellen, müsste neben der veränderten ReportDefinition Klasse mit Filter auch die ursprüngliche ReportDefinition Klasse ohne Filter erhalten bleiben. In der Konfigurationsdatei müsste dann festgelegt werden, wie viel Speicher für Report Definitionen mit und wie viel für Report Definitionen ohne Filter reserviert werden soll. Außerdem müsste ein Mechanismus entworfen werden, der regelt, ob neu erstellte Report Definitionen in einer ReportDefinition Klasse mit oder ohne Filtermöglichkeit gespeichert werden sollen. Eine Möglichkeit wäre, dies über die SID zu lösen, so dass zum Beispiel die SIDs 1-1000 immer in nicht filterbaren und die SIDs über 1000 in filterbaren ReportDefinition Instanzen gespeichert werden.
Das Einlesen neuer Filter Definitionen funktioniert ebenfalls ähnlich wie bei Report Definitionen. Der Algorithmus zum Einlesen befindet sich in der FilterDefinition Klasse selbst, wird allerdings in der Regel vom Report Manager aus aufgerufen. Wird die Einlese-Funktion des Report Managers aufgerufen, sieht dieser Report Manager zunächst nach für welche SID dieser Filter bestimmt ist, sucht die entsprechende Report Definition und ruft die Einlese-Funktion des dazugehörigen Filters auf. Vor jedem Einlesen wird darüber hinaus die „check“-Funktion der Filter Definition Klasse aufgerufen, die die Werte, die eingelesen werden sollen, auf Plausibilität prüft. Um das unbeabsichtigte Überschreiben eines Filters zu vermeiden, prüft die check-Funktion zusätzlich, ob eine eventuell bereits vorhandene Filter Definition mittels „invalidate“ für ungültig erklärt wurde. Ist dies nicht der Fall, wird die alte Definition behalten und die neu einzulesende abgewiesen. Nicht überprüft wird dagegen, ob alle in den Triggern angegebenen Message IDs auch tatsächlich zu filterbaren Messages gehören oder evtl. nur zu regulären RelayMessages. Dies würde die Verwendung von RTTI erfordern, was wie in

Abschnitt 2.4.1 erläutert in eingebetteten Systemen vermieden werden sollte. Es liegt also in der Verantwortung der Bodenkontrolle darauf zu achten, dass lediglich die IDs filterbarer Messages als Trigger Parameter angegeben werden. Dies könnte beispielsweise in Form einer Validierungs-Funktion innerhalb der Telekommando Software der Bodenkontrolle realisiert werden.

Die Erweiterung der DiagnosticReportManager Klasse gestaltet sich relativ simpel. Die neu hinzugefügten Funktionen des Report Managers beschränken sich im Wesentlichen darauf, Anfragen zum Erstellen, Abfragen und Löschen von Filtern an die dazugehörige Report Definition weiterzuleiten, die diese wiederum an den Filter weitergibt. Die eigentliche Verarbeitung dieser Anfragen geschieht in der Filter Definition selbst.

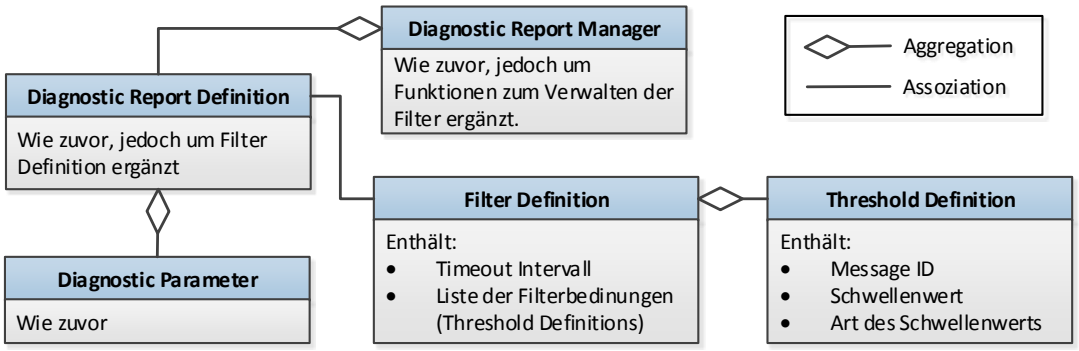


Abb. 12: Filter Definitionen, Filterbedingungen und erweiterter Diagnostic Report Manager

3.3.4 Erweiterter Diagnostic Report Task

Die nötigen Anpassungen des DiagnosticReportTasks betreffen im Wesentlichen die Entscheidungsfindung, ob ein Report generiert werden soll oder nicht. Die eigentliche Generierung des Reports wird nicht geändert. Im ursprünglichen DiagnosticReportTask musste lediglich überprüft werden, ob das Collection Intervall einer Report Definition abgelaufen war und, falls dies der Fall war, der entsprechende Report generiert werden. In der für das Filter Konzept erweiterten Version wird ebenfalls über alle aktiven Report Definitionen iteriert, allerdings muss hier zunächst überprüft werden, ob der zu der Report Definition gehörende Filter aktiv ist. Ist dies nicht der Fall wird, wie zuvor auch, geprüft ob das Collection Intervall erreicht ist und gegebenenfalls ein Report gesendet. Ist der Filter aktiv muss für jeden im Filter enthaltenen Schwellenwert geprüft werden, ob ihre Filter-Bedingung zutrifft und im Falle von mindestens einer wahren Filter-Bedingung der Report generiert und der Timeout Counter zurückgesetzt werden. Trifft auch diese Bedingung nicht zu, muss noch überprüft werden, ob der Report aufgrund

eines abgelaufenen Timeout Counters gesendet werden soll. Ist auch dies nicht der Fall, wird mit der Überprüfung der nächsten Report Definition fortgefahren bzw. nach der letzten Report Definition gewartet bis der DiagnosticReportTask erneut ausgeführt wird.

4 Anwendungsbeispiel

Im vorangegangen Kapitel wurde der Aufbau und die interne Funktionsweise des Filterkonzepts beschrieben. Nun soll die Verwendung des Filtersystems anhand einiger Code-Ausschnitte aus einer Demo-Anwendung näher erläutert werden.

Um eine Message zu erstellen, deren Inhalt als Filter-Kriterium verwendet werden kann, ist es nötig eine von FilterableMessage abgeleitete Klasse zu erstellen. Die Instanziierung des Klassen-Templates muss dabei mit dem für die Message gewünschten Datentyp erfolgen. Möchte man beispielsweise eine Message implementieren, die einen Integer Wert transportiert, müsste die dazugehörige Klasse wie folgt aussehen:

```
1  class IntMessage : public FilterableMessage<int> {
2  public:
3    IntMessage(unsigned const int inputs, Message::Id id):
4      FilterableMessage(inputs, id) { }
5
6    bool trigger(double delta, bool absolute) {
7      bool trigger = false;
8      double oldDouble = static_cast<double>(values[1]);
9      double newDouble = static_cast<double>(values[0]);
10     if(absolute) {
11       trigger = (((oldDouble-newDouble)>=delta)||
12         ((newDouble-oldDouble)>=delta));
13     } else {
14       trigger = (((oldDouble-newDouble)/oldDouble>=delta)||
15         ((newDouble-oldDouble)/oldDouble>=delta));
16     }
17     return trigger;
18   }
19 };
```

Listing 1: Filterbare Message Klasse zum Transport von Integer Werten

Wie bereits im vorherigen Kapitel erwähnt ist es für jede neue, von FilterableMessage abgeleitete Klasse nötig die trigger-Funktion zu überschreiben. Im Falle von Integer Werten ist diese Funktion sehr simpel: Die letzten beiden Werte, die mittels der push-Funktion an die Message übergeben wurden, sind in den Variablen values[0] und values[1] gespeichert. Diese Werte werden zunächst zu einem double Wert umgewandelt, damit sie im nächsten Schritt voneinander subtrahiert und mit dem als double vorliegenden Schwellenwert (delta) verglichen werden können. Da nur der Absolutwert der Abweichung, nicht jedoch das Vorzeichen von Interesse ist, wird einmal der alte vom neuen und einmal der neue vom alten Wert abgezogen und mit dem Schwellenwert verglichen. Außerdem wird mithilfe des „absolute“-Parameters unterschieden, ob es sich bei dem delta-Parameter um einen Schwellenwert für eine absolute oder relative Abweichung handelt. Im Falle einer relativen Abweichung muss die Differenz des neuen und des alten Wertes noch durch den alten Wert geteilt werden, um die prozentuale Abweichung zu erhalten. Sollte es sich bei den in der Message enthaltenen Daten nicht um Zahlen handeln, muss wie bereits in Abschnitt 3.3.2 erwähnt, ein Kriterium gefunden werden, anhand dessen der Datentyp auf einen double Wert reduziert werden kann. Würde die Message beispielsweise einen Vektor enthalten, könnte die Winkeländerung des Vektors zur Normalebene oder zu einer der Achsen des Raumfahrzeugs als Filterkriterium verwendet werden. Der direkte Vergleich zweier Vektoren zueinander ist, wie bereits erwähnt, nicht möglich, da die Abweichung von einem Wert zu einem anderen nur für Zahlenwerte definiert ist.

Möchte man die ebenfalls in Abschnitt 3.3.2 beschriebene Möglichkeit nutzen, eine FIFO Struktur mit mehreren Elementen innerhalb einer FilterableMessage zu erstellen, muss dem Klassen-Template neben dem Datentyp noch die Größe des FIFO Speichers übergeben werden. Für das in Abschnitt 3.1.1 aufgeführte Beispiel der Sensor-Fusion, bei dem die letzten 10 Werte eines Gyroskops bzw. eines Magnetometers gespeichert werden sollen, würde die benötigte Klasse wie folgt aussehen:

```
1  class DoubleFIFOMsg:
2    public FilterableMessage<double, 10> {
3  public:
4    DoubleFIFOMsg(unsigned const int inputs, Message::Id id):
5      FilterableMessage(inputs, id) { }
6
7    bool trigger(double delta, bool absolute) {
8      bool trigger = false;
9      if(absolute) {
10       trigger = (((values[1]-values[0])>=delta)||
11         ((values[0]-values[1])>=delta));
```

```
12      } else {
13        trigger = (((values[1]-values[0])/values[1]>=delta)||
14          ((values[0]-values[1])/values[1]>=delta));
15      }
16      return trigger;
17    }
18  };
```

Listing 2: Filterbare Message mit FIFO

Mittels der push-Funktion können nun beliebige Werte vom Typ double übermittelt werden, wobei immer die 10 zuletzt gepushten Werte in dem FIFO enthalten sind. Verbindet man nun eine Message vom Typ DoubleFIFOMessage mit einem TaskInput der so konfiguriert ist, dass er den dazugehörigen Task nach 10 push-Aufrufen aktiviert, lässt sich sehr einfach eine Vorverarbeitung der analogen Sensorwerte, wie in dem Sensor-Fusion Beispiel beschrieben, realisieren. Der Task, der diese Vorverarbeitung vornehmen soll, kann dann die letzten 10 Werte mittels der Funktion „getData“ abfragen, wobei die Funktion als Argument die Position des gewünschten Werts, also in diesem Fall eine Zahl zwischen 0 und 9, entgegen nimmt. In der trigger-Funktion könnten neben den Variablen values[0] und values[1], die die letzten beiden Werte enthalten, theoretisch auch die anderen 8 Werte mittels values[2] bis values[9] abgefragt werden. In der Regel ist es jedoch am sinnvollsten, für eine Filterbedingung lediglich die beiden zuletzt gepushten Werte zu vergleichen.

Um die erstellten Message Klassen auch nutzen zu können, müssen Instanzen dieser Klassen erstellt werden (vgl. Listing 3, Zeile 1). Außerdem müssen die Messages mit der angegebenen Anzahl an Inputs verknüpft werden und den Inputs ein Task zugeordnet werden (vgl. Listing 3, Zeilen 3 - 13). Da die execute-Funktion der Task Klasse rein virtuell ist, musste für die Demo-Anwendung eine von Task abgeleitete Klasse „Dummy-Task“ erstellt werden, die zwar keinerlei Funktion hat, jedoch eine leere Instanziierung der execute-Funktion bietet, so dass ein Objekt dieser Klasse erstellt werden kann. Des Weiteren muss eine Instanz des DiagnosticReportManagers sowie eine Instanz des DiagnosticReportIterators mit dem zuvor erstellten Manager als Argument erstellt werden.

```
1 // IntMessage mit einem Input und der ID TEST_DUMMY1
2 IntMessage intMsg(1, Message::TEST_DUMMY1);
3
4 // DummyTask mit einem Input
5 DummyTask task(1);
6
```

```
// TaskInput mit aktivierungs Threshold von 1
Tasking::TaskInput input(1);

// Verknuepfe Input mit Message
input.associate(&intMsg);

// Verknuepfe Task mit Input (Input Key = 0)
task.add(0, &input);

// DiagnosticReportManager und Iterator
DiagnosticReportManager manager;
DiagnosticReportIterator it(manager);
```

Listing 3: Erstellung und Verknüpfung neuer Messages, TaskInputs und Tasks sowie eines DiagnosticReportManagers

Über den Report Manager können dann neue Report Definitionen und Filter eingelesen werden (vgl. Listing 4). Zum Einlesen neuer Report Definitionen ist es nötig, ein ausreichend großes Feld als Zwischenspeicher für den Serialisierer zu erstellen. In diesem Feld werden die Daten zum Übermitteln an den Manager zwischengespeichert. Ist die Report Definition erfolgreich eingelesen, wird es nicht weiter benötigt. Nachdem das Serialisierer-Objekt erstellt wurde, kann begonnen werden, die Daten der Report Definition in den Zwischenspeicher zu schreiben. Es ist wichtig die im PUS Standard vorgegebene Telekommando Struktur einzuhalten, d.h. es muss immer zuerst die SID, gefolgt von Collection Interval, Anzahl der Parameter sowie ebenso vieler Parameter IDs angegeben werden. Als Parameter ID können die IDs aller von RelayMessage abgeleiteten Messages sowie aller Parameter verwendet werden. Gehört die ID zu einer von RelayMessage abgeleiteten Klasse muss zusätzlich ein „Marker“ hinzuaddiert werden, damit das Housekeeping System entscheiden kann, wo nach der entsprechenden ID gesucht werden soll. Der letzte Wert der Report Definition (NFA) steht für „Number of Fixed-Length Arrays“ und sollte immer 0 sein, da das Diagnostic Report System die entsprechende Funktion nicht unterstützt. Zum Schluss muss die Report Definition noch über den DiagnosticReportManager eingelesen sowie bei Bedarf aktiviert werden.

```
uint8_t dprData[40]; // Zwischenspeicher
Serializer::Serialize serialize(dprData);

serialize.store32(1u); // SID
serialize.store32(1u); // Collection Intervall
serialize.store32(1u); // Anzahl der Parameter
```

```
serialize.store32(RelayMessage::marker +
    Message::TEST_DUMMY1); //  ID der intMsg + Marker
serialize.store32(0u); // NFA

manager.read(dprData); // Einlesen der Report Definition
                       // in das Housekeeping System

manager.enable(1u); // Aktivieren des Reports mit SID 1
```

Listing 4: Erstellung neuer Report Definitionen

Um die zyklische Generierung der Housekeeping Reports zu aktivieren, muss lediglich noch das Tasking Framework aktiviert werden:

```
// Initialisierung und Start des Tasking Frameworks
Tasking::initialize();
Tasking::start();
```

Listing 5: Initialisierung und Start des Tasking Frameworks

Ab diesem Zeitpunkt wird regelmäßig ein Report gemäß der zuvor erstellten Report Definition generiert und ausgegeben. Möchte man nun die Filter Funktion des Housekeeping and Diagnostic Data Reporting System nutzen, ist es nötig, noch eine Filter Definition für die zu filternde Report Definition zu erstellen. Dies funktioniert ähnlich wie die Erstellung neuer Report Definitionen:

```
uint8_t dprData[40]; // Zwischenspeicher
Serializer::Serialize serialize(dprData);

serialize.store32(1u); // SID
serialize.store32(50u); // Timeout Intervall
serialize.store32(1u); // Anzahl der Trigger
serialize.store32(Message::TEST_DUMMY1); // ID des Triggers
serialize.storeBool(true);  // Schwellenwert ist absolut
serialize.storeDouble(2.0); // Schwellenwert ist 2.0

manager.readFilter(dprData); // Einlesen des Filters
```

Listing 6: Erstellung eines Filters

Eine Filter Definition muss im Gegensatz zu einer Report Definition nicht separat aktiviert werden. Sie kann, wie im PUS Standard vorgesehen, lediglich erstellt und gelöscht werden. Der erste Wert, der mithilfe des Serialisierers in den Zwischenspeicher geladen wird, ist wie auch schon bei den Report Definitionen die SID. Im Anschluss daran werden Timeout Intervall, die Anzahl der Trigger sowie für jeden Trigger eine ID, die Art des Schwellenwerts sowie der Schwellenwert selbst übergeben. Anders als bei Report Definitionen, dürfen bei Filter Definitionen lediglich die IDs von Messages, die von FilterableMessage abgeleitet sind, angegeben werden, nicht jedoch die von RelayMessages oder Parametern. Der in Listing 5 definierte Filter generiert den Report mit der SID 1, wenn sich der Wert der beiden letzten Zahlen, die mittels der push-Funktion an die Message „intMsg“ übergeben wurde, um mehr als 2 unterscheidet. Verglichen und gegebenenfalls gesendet wird dabei immer im Basistakt des Housekeeping & Diagnostic Report Systems - in den Demo- und Test-Anwendungen also beispielsweise alle 200 Millisekunden.

5 Test und Evaluierung

5.1 Unit-Tests

Wie bereits in Abschnitt 3.2 erwähnt, soll die Entwicklung des Filter Konzepts dem Test-Driven-Development Ansatz (siehe Unterkapitel 2.4.2) folgen. Als Test-Framework wird dabei das „Google C++ Testing Framework“, häufig auch „Google Test“ oder „gTest“ genannt, eingesetzt. Google Test basiert auf der beliebten xUnit Architektur und kommt bei zahlreichen bekannten Projekten wie beispielsweise dem Chromium Projekt oder dem LLVM Compiler zum Einsatz. [22]

Auch bei der bereits vorhandenen Implementierung des Housekeeping & Diagnostic Report Systems kam Google Test als Test-Werkzeug zum Einsatz, weshalb die Tests der wiederverwendeten und erweiterten Klassen nahtlos in die Tests der neuen Klassen integriert werden konnten. Da es den Rahmen dieser Arbeit überschreiten würde jeden Test einzeln zu erläutern, wird im Folgenden lediglich kurz auf den Aufbau und die Vorgehensweise bei der Erstellung der Tests eingegangen. Prinzipiell wurde für jede Klasse ein eigener „Test Case“ geschrieben, der wiederum mehrere Tests enthält. Die Test Cases aller Klassen werden beim Kompilieren in ein Test-Programm zusammen gefasst, das der Reihe nach die Tests aller Test Cases ausführt. Das Test-Programm kann in der Konsole ausgeführt werden und gibt das Ergebnis der einzelnen Tests sowie eventuelle Fehler- und Debug-Meldungen aus. Generell wurde für jede Funktion einer Klasse (außer bei simplen Getter- und Setter-Funktionen) ein eigener Test geschrieben, der die Rückgabewerte oder sonstige, von der Funktion ausgelösten Ereignisse bei ver-

[22] https://code.google.com/p/googletest/

schiedenen Eingaben überprüft. Dabei wurde stets versucht neben Standard-Eingaben auch Grenz- und Sonderfälle zu berücksichtigen. Darüber hinaus wurden in manchen Test Cases übergeordnete Tests geschrieben, die nicht nur einzelne Funktionen, sondern das Zusammenspiel verschiedener Funktionen innerhalb dieser Klasse überprüfen. Die Testfälle liegen dem Sourcecode bei und sind beliebig reproduzierbar. Es kann daher davon ausgegangen werden, dass das Konzept wie gewünscht arbeitet und den Anforderungen entspricht.

5.2 Profiling-Tests

5.2.1 Überblick

Um die Leistungsfähigkeit des Konzepts zu prüfen, wurde der Prototyp zusammen mit dem Tasking Framework in eine statische Bibliothek kompiliert, so dass es schnell und einfach in verschiedenen Szenarien getestet werden kann.
Wichtig ist zu beachten, dass in diesem Unterkapitel mit „Test“ und „Testfälle“ die Test-Programme gemeint sind, die zum Zwecke des Profiling geschrieben wurden, nicht etwa die zuvor erwähnten Unit-Tests zum Überprüfen der Funktionalität eines Moduls. Jedes Testszenario wurde durch ein eigenes Test-Programm realisiert, das ähnlich wie das in Kapitel 4 angesprochene Demo-Programm aufgebaut ist. Die Test-Programme orientieren sich damit bezüglich der Verwendung des Diagnostic & Housekeeping Report System an der späteren Onboard-Software, sind ansonsten jedoch natürlich stark vereinfacht. Um diverse Eigenschaften der Programme wie die Laufzeit einzelner Funktionen oder den Bedarf an Heap- und Stack-Speicher zu messen, wurde das Valgrind Toolset verwendet. Valgrind enthält eine Reihe von Profiling Werkzeugen die für verschiedene Zwecke eingesetzt werden können. Konkret kamen beim Profiling des Filter Konzepts die Werkzeuge „Massif“ zum Überwachen des Heap- und Stack-Speichers sowie „Callgrind“ zum Ermitteln der Laufzeit und der Anzahl der Aufrufe einzelner Funktionen zum Einsatz. Die Fragen, denen mithilfe des Profiling nachgegangen werden soll, sind:

- Wie hoch ist der Bedarf an Stack-Speicher und bleibt er während der Laufzeit des Programms ungefähr gleich oder sind große Spitzen an Speicherbedarf zu beobachten?
- Wie hoch ist der Bedarf an Heap-Speicher und wird, abgesehen von der Initialisierungsphase, tatsächlich kein (dynamischer) Heap Speicher zugewiesen?
- Steigt der Speicherbedarf linear wenn ein einzelner Parameter, wie beispielsweise die Anzahl maximaler Report Definitionen, geändert wird oder steigt er schneller (z.B. exponentiell)?

- Wenn ein einzelner Parameter geändert wird: Wie verändert sich das Laufzeitverhalten des Programms? Insbesondere die Laufzeit der häufig aufgerufenen Funktionen wie der execute Funktion des DiagnosticReportTasks und der push-Funktionen verschiedener Message Arten sind zu untersuchen.

Da die Fragen grob in die zwei Gruppen „Speicher“ und „Laufzeit“ unterteilt werden können, wurden auch die Testfälle dementsprechend aufgeteilt. Die beiden Folgenden Unterkapitel sollen eine kurze Übersicht über die durchgeführten Tests sowie eine Zusammenfassung und Interpretation der erhaltenen Ergebnisse liefern.

5.2.2 Laufzeit

Im Rahmen der Laufzeittests wurden die Anzahl der maximal möglichen Report Definitionen, die der maximalen Threshold Definitionen pro Filter sowie die Anzahl der aktiven Report Definitionen, aktiven Filter, aktiven Threshold Definitionen und die Größe und Anzahl der Messages variiert. Im Anschluss wurde die Laufzeit der execute-Funktion des DiagnosticReportTasks sowie die push-Funktion der Messages analysiert, wobei es in den Tests zwei verschiedene Message Arten gab. Zum einen wurde eine simple Integer-Message, ähnlich wie die der Demo-Anwendung aus Kapitel 4 implementiert, zum anderen gab es eine „Vector“-Message, die zum Testen großer Datenstrukturen ein Integer-Array mit Größen zwischen 100 und 10000 enthielt. Zusätzlich zu den zuvor genannten Funktionen wurde bei jedem Test die Laufzeit einer Funktion namens „calc_delay“ gemessen, die zu Vergleichszwecken einige simple Rechenoperationen durchführte. Die Laufzeit der calc_delay-Funktion war bei allen Tests konstant, was bedeutet, dass die erhaltenen Werte der übrigen Funktionen miteinander vergleichbar sind. Darüber hinaus wurden bei einigen Testfällen auch die Laufzeiten einzelner Unterfunktionen untersucht, um die Ursache eines Laufzeitunterschieds genauer ermitteln zu können. Im Folgenden werden kurz die einzelnen Tests erläutert und die wichtigsten Erkenntnisse daraus genannt. Das vollständige Testprotokoll findet sich in Anhang C.

Beim ersten Test wurde sowohl die Anzahl maximal möglicher Report Definitionen als auch die Anzahl maximal möglicher Threshold Definitionen variiert. Wie erwartet brachte eine Erhöhung der Werte keinerlei Auswirkungen auf die Laufzeit der push-Aufrufe. Die Laufzeit der execute-Funktion blieb bei Erhöhung der maximalen Threshold Definitionen ebenfalls konstant, bei Erhöhung der maximal möglichen Report Definitionen stieg sie leicht. Eine genauere Betrachtung der Laufzeit der Unterfunktionen brachte hervor, dass der Laufzeitunterschied lediglich durch das häufigere Durchlaufen einer sehr kurzen Schleife in der next-Funktion des DiagnosticReportIterators verursacht wird und somit unbedenklich ist.

Die Veränderung der Anzahl aktiver Report Definitionen im zweiten Test brachte erneut keine Veränderung der push-Laufzeiten mit sich, dafür jedoch eine Erhöhung der Laufzeit der execute-Funktion. Dies hat den Grund, dass bei einer größeren Anzahl aktiver Report Definitionen folglich auch über mehr Definitionen iteriert und mehr Reports gesendet werden müssen. Der gemessene Anstieg der Laufzeit in Abhängigkeit aktiver Report Definitionen ist jedoch nach einer graphischen Auswertung als linear einzuschätzen und daher akzeptabel. Eine genauere Untersuchung ergab zudem, dass ein Großteil der Laufzeitdifferenz durch das häufigere Aufrufen der „sendData“-Funktion des Debug-Interfaces entstand und somit nicht relevant für die Leistungsbewertung des Filter-Konzepts ist.

Im dritten Test wurde die Anzahl der aktiven Filter Definitionen schrittweise erhöht. Auch bei diesem Test blieb die Laufzeit der push-Funktionen unverändert. Die Laufzeit der execute-Funktion ging mit zunehmender Zahl aktiver Filter zurück, da die zu filternden Reports aufgrund der Filterbedingungen seltener generiert wurden. Die Laufzeit pro gesendetem Report blieb jedoch erwartungsgemäß konstant.

Bei Test 4 wurde das Verhalten bei zunehmender Anzahl an Triggern (Threshold Definitionen) pro Filter Definition untersucht. Es wurden insgesamt drei Messungen durchgeführt wobei bei jeder Messung ein Trigger hinzugefügt wurde. Es entstand ein geringer Laufzeitunterschied in der execute-Funktion des DiagnosticReportTasks, der dadurch entstand, dass mehr Filterbedingungen pro Filter überprüft werden mussten. Die Größe des Laufzeitunterschieds wird maßgeblich davon beeinflusst, wie lange die Überprüfung der einzelnen Filterbedingungen braucht und ist somit vom Datentyp der Message abhängig. Da bei den Tests lediglich Messages mit ähnlich kurzen isTriggered-Funktionen verwendet wurden, war der Laufzeitanstieg der execute-Funktion konstant. Die Laufzeit der übrigen überwachten Funktionen änderte sich nicht.

In den Tests 5 und 6 wurde die Anzahl der verschiedenen Message Arten sowie die Größe der Vector-Message variiert. Die Vergrößerung des in Vector-Message enthaltenen Arrays brachte aufgrund der Zuweisungsoperatoren innerhalb der push-Funktion einen linearen Zuwachs der Laufzeit von push mit sich. Da die Vector-Message jedoch lediglich als Trigger verwendet wurde und in keinem Report enthalten war, änderte sich die Laufzeit der execute-Funktion nicht. Eine Erhöhung der Anzahl der FilterableMessages brachte aufgrund der häufiger aufgerufenen push-Funktionen kumulativ betrachtet ebenfalls eine Erhöhung der push-Laufzeit mit sich. Die Laufzeit pro Aufruf änderte sich dagegen nicht. Die Erhöhung der Anzahl der in Report Definitionen enthaltenen Messages verursachte erwartungsgemäß eine etwas längere Laufzeit der execute-Funktion, da mehr Daten abgefragt und übertragen werden mussten. Jedoch skalierten auch hier wieder alle Laufzeitverzögerungen linear mit ihrer Ursache, wes-

halb sie als ungefährlich eingestuft werden können.

Insgesamt verhält sich der Prototyp des Filter-Konzepts bezüglich des Laufzeitverhaltens wie erwartet. Eine Leistungsbewertung ist angesichts der mangelnden Vergleichsmöglichkeiten nur schwer möglich. Die Tatsache, dass die Laufzeit im schlimmsten Fall lediglich linear zu den veränderten Parametern steigt, deutet jedoch darauf hin, dass das Filter-Konzept relativ effektiv arbeitet.

5.2.3 Speicherbedarf

Die Profiling-Tests zur Ermittlung des Speicherbedarfs des Prototypen sind ähnlich aufgebaut wie die Laufzeit-Tests. Variiert wurden erneut die maximale Anzahl an Filter Definitionen, die maximale Anzahl an Threshold Definitionen sowie die Größe der verwendeten Messages mithilfe der Vector-Message-Klasse. Außerdem wurden im Rahmen des Speicher-Profiling noch eine Reihe weiterer Parameter wie Anzahl und Art der Messages, Anzahl aktiver Report Definitionen und Anzahl aktiver Threshold Definitionen verändert, die jedoch alle – wie erwartet – ohne Auswirkungen auf den Speicherbedarf blieben. Gemessen wurde jeweils der zeitliche Verlauf des benötigten Stack- und Heap-Speichers sowie die Größe von Text, Data und BSS. Der Aufbau des virtuellen Speicherbereichs eines Prozesses ist in Abbildung 13 dargestellt. Im Text-Bereich ist der Code gespeichert, Data enthält alle Konstanten und BSS die globalen und statischen Variablen. Auf dem Stack werden lokale Variablen abgelegt, der Heap-Speicher enthält dynamisch belegten Speicher. Stack und Heap sind die einzigen Speicherbereiche, die sich während der Ausführung des Prozesses verändern, wobei der Stack bei der größten Speicheradresse beginnt und zu kleineren Adressen wächst, während der Heap Speicher direkt nach dem BSS-Bereich beginnt und zu größeren Adressen wächst. Im Rahmen des Tests wurde zum einen der maximal benötigte Stack- und Heap-Speicher ermittelt, zum anderen wurden bei einigen Tests auch Diagramme, die den zeitlichen Verlauf der Speicherbelegung zeigen, angefertigt. Exemplarisch für den Verlauf der Stack-Speicherbelegung ist in Abbildung 14 das Diagramm des ersten Tests dargestellt, bei dem die Anzahl der maximal möglichen Threshold Definitionen zwischen 25 und 100 variiert wurde. Wie zu erkennen ist, ist zwar die Höhe der Speicherbelegung unterschiedlich, der Verlauf jedoch sehr ähnlich.

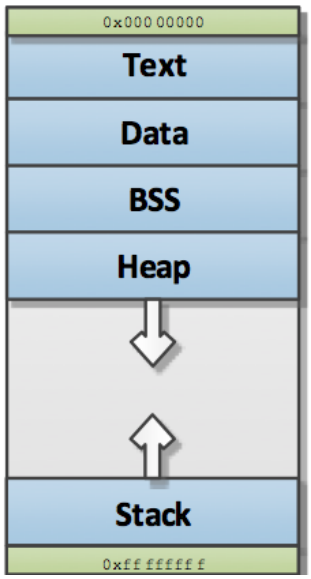


Abb. 13: Struktur eines virtuellen Speicherbereichs[23]

[23] Zeichnung nach Raghavan, [RLN05] [S.344, Figure 10.1]

Nach einer kurzen Initialisierungsphase ist bereits ein Großteil des benötigten Stack-Speichers belegt, im weiteren Verlauf sind lediglich geringe Spitzen und Tiefpunkte erkennbar. Auch beim zweiten Test, bei dem die Anzahl der maximal möglichen Report Definitionen erhöht wurde, war der Verlauf des Stack-Speicherbedarfs ähnlich. Die Maximalwerte des Speicherbedarfs skalierten jeweils linear mit den angegebenen Maxima für Threshold und Report Definitionen. Der benötigte Heap-Speicher war bei allen Tests konstant und wurde, wie erwartet, lediglich in der Initialisierungsphase des Programms belegt. Auch die übrigen Werte Text, Daten und BSS blieben im Verlauf der Tests nahezu konstant. Beim letzten Test des Speicher-Profiling wurde zu den bisher verwendeten Integer-Messages eine Vector-Message mit 1000 Integer Werten sowie ein Objekt vom Typ Vector hinzugefügt, das mittels der push-Funktion an die Vector-Message übergeben wurde. Wie erwartet erhöhte sich der maximale Bedarf an Stack-Speicher gegenüber dem vorherigen Test um ungefähr $3 \cdot 4000$ Byte, da für den Vector 4000 Byte und für die Vector-Message $2 \cdot 4000$ Byte benötigt wurden.
Wie auch schon bei den Laufzeittests lässt sich der Speicherbedarf des Filter-Konzepts aufgrund der fehlenden Vergleichsmöglichkeiten schwer bewerten. Ein Großteil des Speicherbedarfs entstand jedoch durch die Reservierung des Speichers für Report- und Threshold-Definitionen, die unabhängig vom Design des Filter-Konzepts unumgänglich sind. Die Anforderung, abgesehen von der Initialisierungsphase keinen Heap-Speicher dynamisch zu belegen wurde ebenfalls erfüllt. Da keine großen Spitzen im Verlauf der Stackspeicherbelegung zu erkennen sind, ist auch die Gefahr eines Speicherüberlaufs durch das Filter-Konzept unwahrscheinlich.

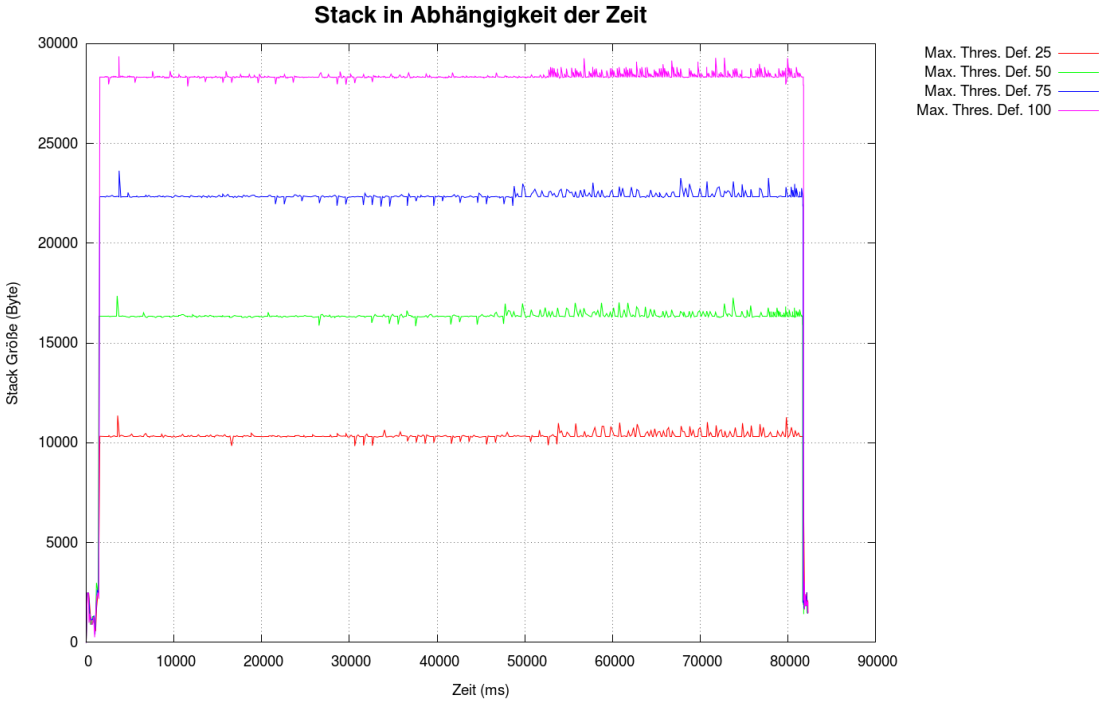


Abb. 14: Verlauf der Stackspeicherbelegung während des ersten Speicher-Tests

6 Fazit und Ausblick

Durch das im Rahmen dieser Arbeit entwickelte Filter-Konzept verfügt das DLR nun über einen Entwurf, auf dessen Grundlage ein Filter-System gemäß des ECSS Standards umgesetzt werden kann. Auch wenn das Konzept aufgrund der Eigenschaften des zugrunde liegenden Tasking Frameworks in einigen Punkten von den im Standard vorgesehenen Spezifikationen abweicht, wurden die grundlegenden Funktionen und Eigenschaften des Filters, wie in der Aufgabenstellung gefordert, umgesetzt. Da sowohl beim Entwurf des Konzepts als auch bei der Implementierung des Prototyps auf gängige Regeln und Richtlinien für die Entwicklung sicherheitskritischer Software Rücksicht genommen wurde, sollte es problemlos möglich sein, das Konzept oder sogar Teile des Prototyps in reale Onboard-Software zu übertragen. Die abschließenden Profiling-Tests bestätigten zum einen die korrekte Funktionsweise des Prototyps, zum anderen ermöglichten sie eine grobe Abschätzung der Leistungsfähigkeit des Konzepts. Auch wenn es ohne entsprechende Vergleichsmöglichkeiten schwierig ist konkrete Aussagen über die Leistungsfähigkeit zu machen, kann aufgrund der Tests angenommen werden, dass sich sowohl der benötigte Speicher als auch die in Anspruch genommene Rechenzeit in einem guten Rahmen befinden.

Um das Filter-Konzept nun auch tatsächlich verwenden zu können, muss in einem weiteren Schritt geprüft werden, bei welchen laufenden und geplanten Missionen das Konzept sinnvoll angewendet werden kann und welche Anpassungen am Konzept und an der Onboard Software des Raumfahrzeugs nötig sind. Außerdem müssen, in Abhängigkeit der jeweiligen Mission, sinnvolle Werte für die einstellbaren Parameter des Filter-Systems ermittelt werden. Darüber hinaus könnte evaluiert werden, ob sich das Filter-Konzept unter Umständen sinnvoll mit anderen, für die Mission benötigten Services, kombinieren ließe. So fielen beispielsweise bei einem ersten Vergleich der Filter-Subservices des „Housekeeping and Diagnostic Data Reporting Service“ mit dem in dieser Arbeit nicht näher behandelten „Onboard Monitoring Service“ eine Menge Gemeinsamkeiten auf. Es könnte daher sowohl für die Ressourcen des Onboard Computers als auch für den Arbeitsaufwand der Implementierung des „Onboard Monitoring Service“ von Vorteil sein, diese Services miteinander zu verknüpfen, weshalb diese Option – sollte der „Onboard Monitoring Service“ benötigt werden – auf jeden Fall in Betracht gezogen werden sollte.

Literaturverzeichnis

[ccs95] Consultative Committee for Space Data Systems: *Packet Telemetry (CCSDS Blue Book, 102.0-B-4)*. 1995

[ecs03] European Cooperation for Space Standardization: *ECSS-E-70-41A: Ground systems and operations - Telemetry and telecommand packet utilization*. 2003

[Eic11] Eickhoff, Jens: *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*. Springer, 2011

[esa00] European Space Agency: *C and C++ Coding Standards*. 2000

[LWH09] Ley, Wilfried ; Wittmann, Klaus ; Hallmann, Willi: *Handbook of Space Technology*. Bd. 22. John Wiley & Sons, 2009

[Mad10] Madeyski, Lech: *Test-Driven Development: An Empirical Evaluation of Agile Practice*. Springer, 2010

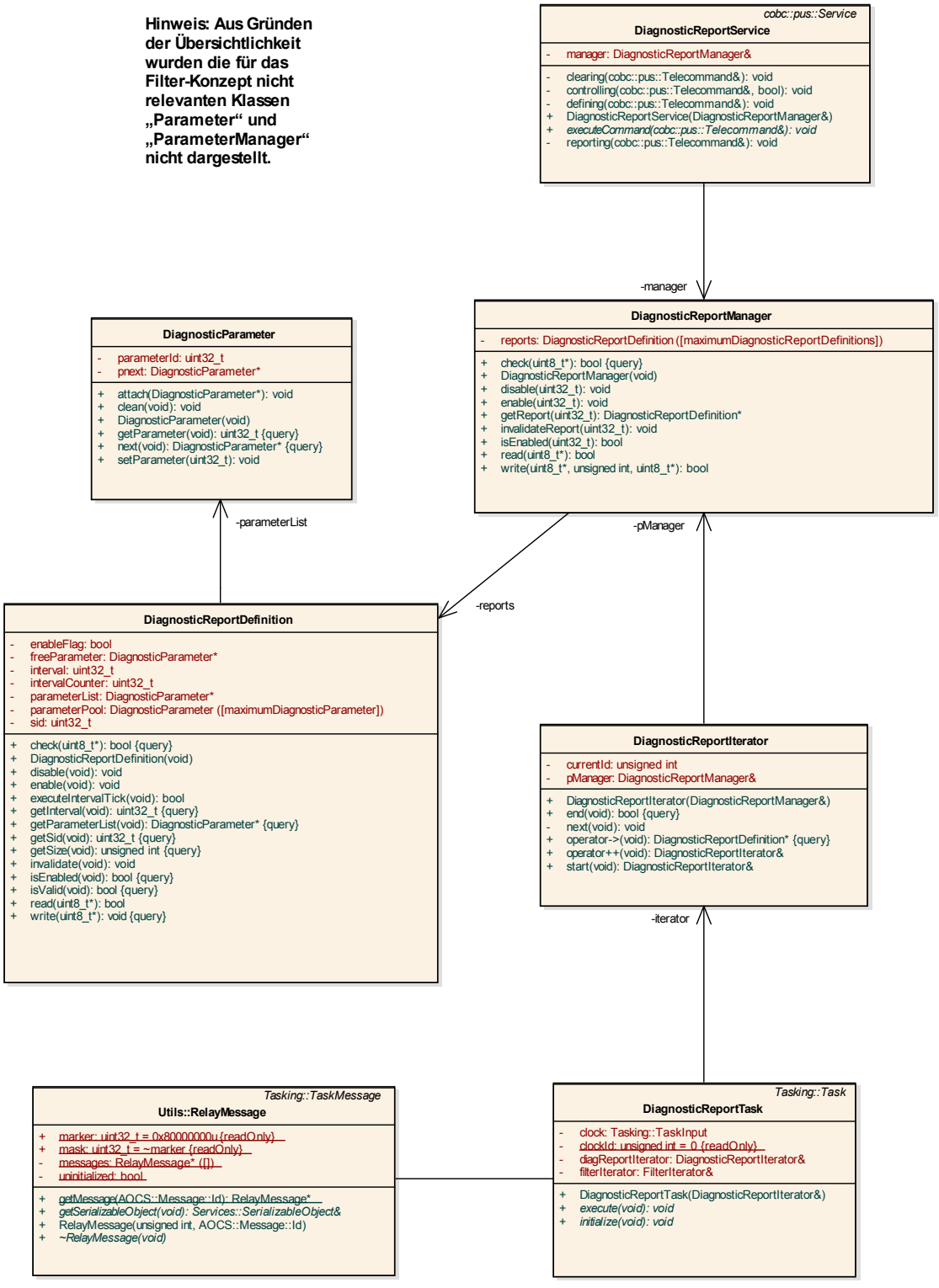
[mis08] The Motor Industry Software Reliability Association: *MISRA C++:2008, Guidelines for the use of the C++ language in critical systems*. 2008

[nas03] Goddard Space Flight Center, Flight Software Branch: *C++ Coding Standard (Code 582)*. 2003

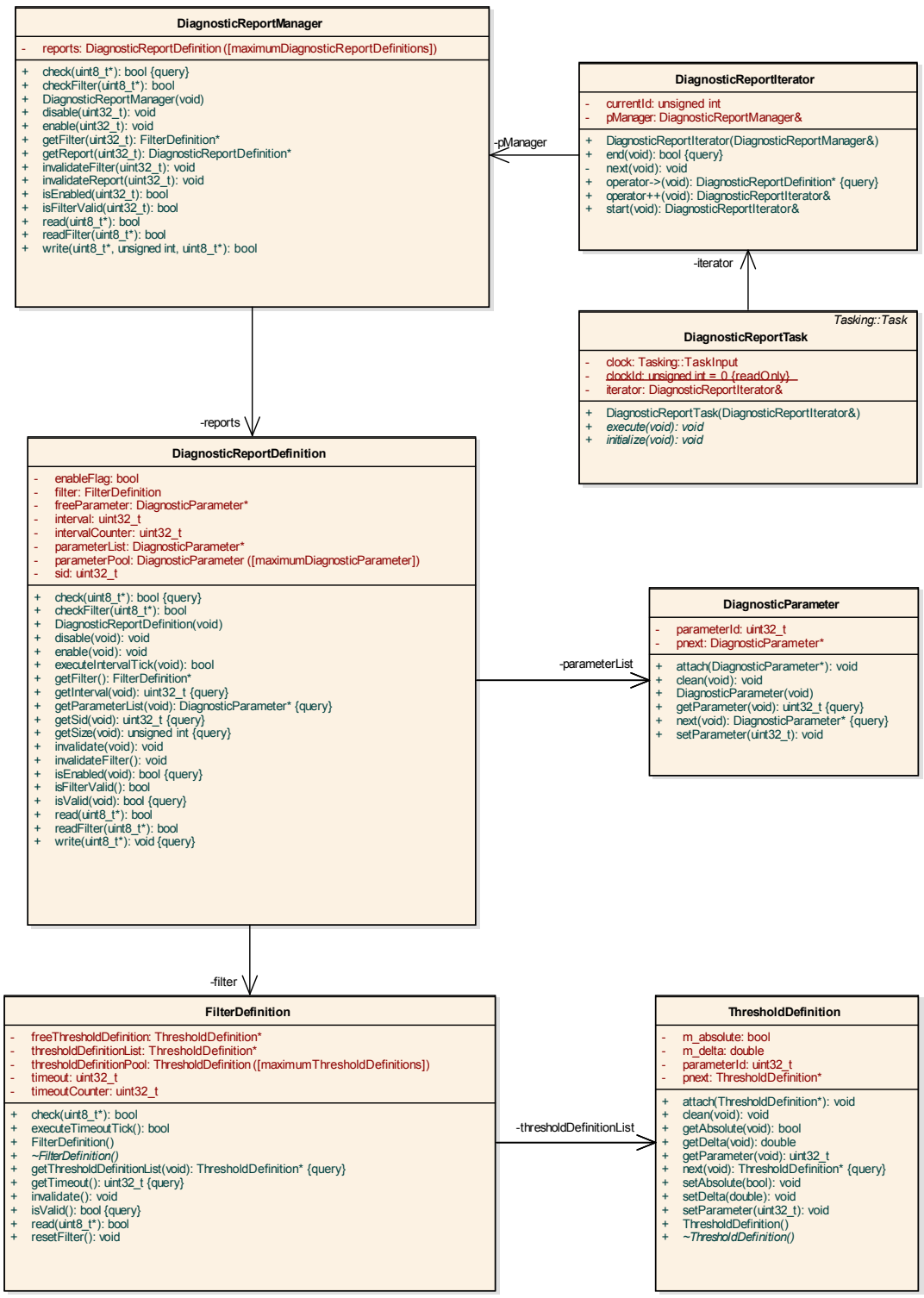
[RLN05] Raghavan, Pichai ; Lad, Amol ; Neelakandan, Sriram: *Embedded Linux system design and development*. CRC Press, 2005

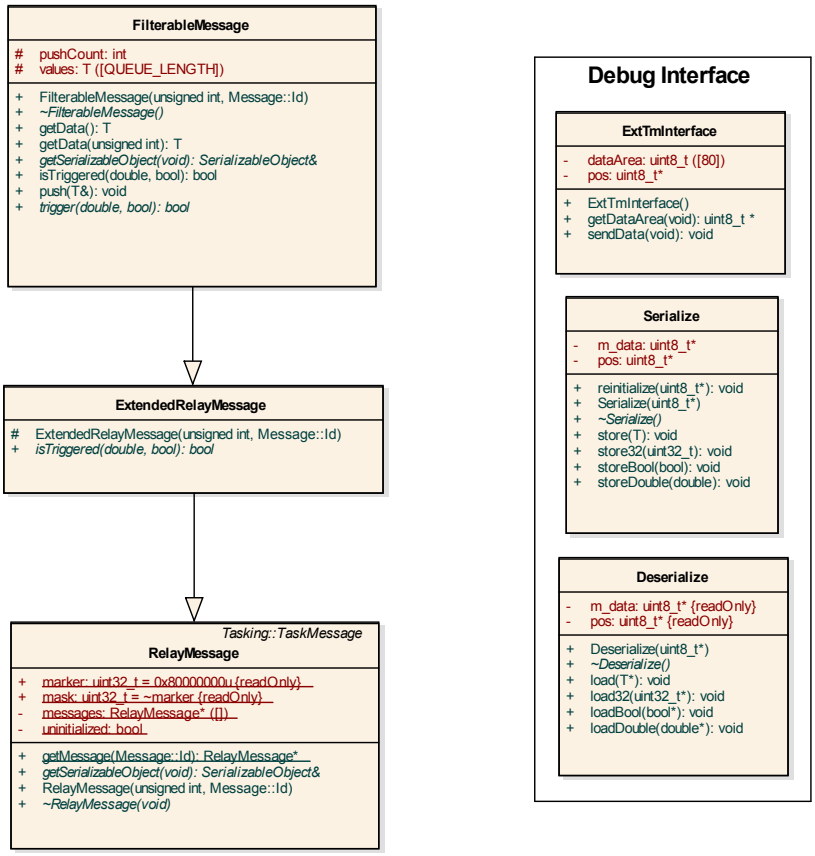
Anhang

A: Ursprüngliche Diagnostic Report Implementierung



B: Filter-Konzept





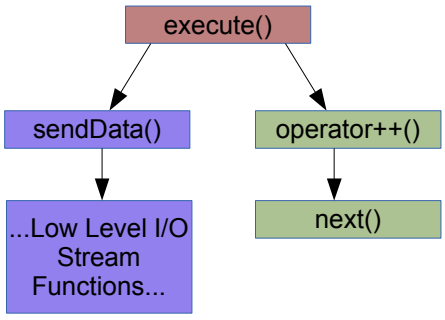
Hinweis: Aus Gründen der Übersichtlichkeit wurden die für das Filter-Konzept nicht relevanten Klassen „Parameter“ und „ParameterManager“ nicht dargestellt.

C: Profiling Protokoll

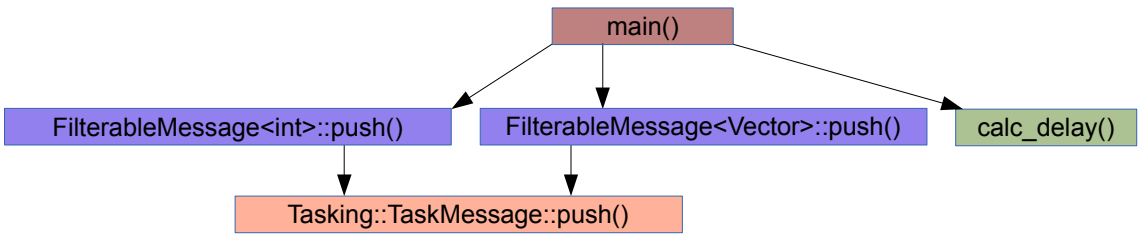
Variablen/Konstanten, die im Rahmen der Tests variiert wurden:

- Anzahl maximal möglicher Report Definitionen (maxDiagnosticReportDefinitions)
- Anzahl maximal möglicher Trigger (maxThresholdDefinitions)
- Aktivierte Report Definitionen sowie Anzahl und Art der Messages pro Report
- Aktivierte Filter sowie Trigger (Threshold Definitionen) pro Filter
- Anzahl und Art der FilterableMessages (im Laufe des Programms gepusht oder nicht gepusht?)
- Anzahl und Art sonstiger Messages

Call Trees:
Alle relevanten Unterfunktionen der execute-Methode



Alle relevanten Unterfunktionen der main-Methode



Anmerkung:
Sofern nicht explizit angegeben sind alle Laufzeitmessungen in CPU Ticks und alle Speichermessungen in Byte zu verstehen.

Laufzeittest 1:
Beschreibung: Im Testprogramm werden zunächst die Anzahl maximaler Trigger (ThresholdDefinition) pro Filter erhöht, dann die Anzahl maximaler Report Definitions, am Ende beides. Bei maxThresholdDefinitions wird (bis auf in der Initialisierungsphase) kein Laufzeitunterschied erwartet. Bei maxDiagnosticReportDefinitions wird lediglich ein geringer Laufzeitunterschied erwartet, der durch die höhere Anzahl an Schleifendurchläufen im Iterator (bzw. in der „next"-Funktion des Iterators) verursacht wird.

MaxDiagnosticReport Definitions	x
MaxThresholdDefinitions	y
Aktive Report Definitions & Anzahl/Art der Messages	1 Report mit 1 IntMessage und 1 DummyMessage (Größe 8u)
Aktive Filter + Trigger pro Filter	1 Filter mit 1 Trigger (IntMessage)
Anzahl und Art von Filterable Messages (gepusht/nicht gepusht)	1 IntMessage (gepusht) und 1 VectorMessage (Größe 4000u, gepusht)
Anzahl und Art sonstiger Messages	1 DummyMessage (Größe 8u)

Messung:

x	10	10	10	50	100	100
y	10	50	100	10	10	100
push(int)	13800	13800	13800	13800	13800	13800
push(Vector)	94480	94480	94480	94480	94480	94480
execute	1259085	1259085	1259085	1729485	2317485	2317486
calc_delay	96720	96720	96720	96720	96720	96720
operator++ (Incl.)	118188	118188	118188	588588	1176588	1176588
operator++ (Self)	7252	7252	7252	7252	7252	7252
sendData	923733	923733	923733	923733	923733	923733

Fazit: Der Tabelle kann entnommen werden, dass maxThresholdDefinitions keinerlei Auswirkungen auf die Laufzeit hat, während maxDiagnosticReportDefinitions die Laufzeit von execute leicht erhöht. Diese Erhöhung wird einzig durch den Aufruf des ++ Operators des DiagnosticReportIterators hervorgerufen. Anhand der „Self-Laufzeit" (Zeit, die tatsächlich in der Funktion selbst verbracht wurde, ohne Unteraufrufe) sieht man jedoch, dass die Erhöhung nicht vom ++ Operator kommen kann, sondern von einer Unterfunktion verursacht werden muss. Da die Funktion next() die einzige Funktion ist, die in dem ++ Operator aufgerufen wird, muss sie der Verursacher der Laufzeitdifferenz sein. Dies deckt sich mit der anfänglichen Vermutung.

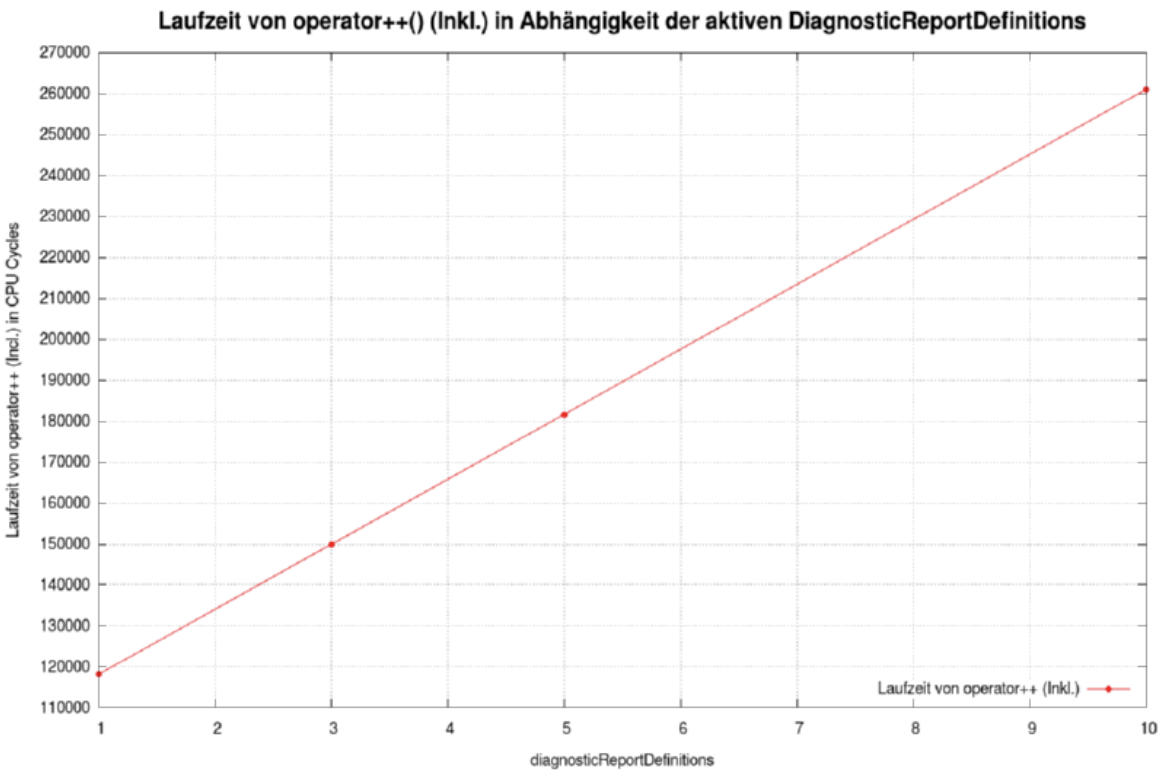
Laufzeittest 2:
Beschreibung: Im Testprogramm wird die Anzahl an aktiven (enabled) DiagnosticReportDefinitions erhöht. Es wird erwartet, dass diese Erhöhung lediglich Auswirkungen auf die Laufzeit von execute hat. Die Erhöhung der Laufzeit sollte jedoch linear zur Zunahme der aktiven DiagnosticReportDefinitions verlaufen und somit unbedenklich sein.

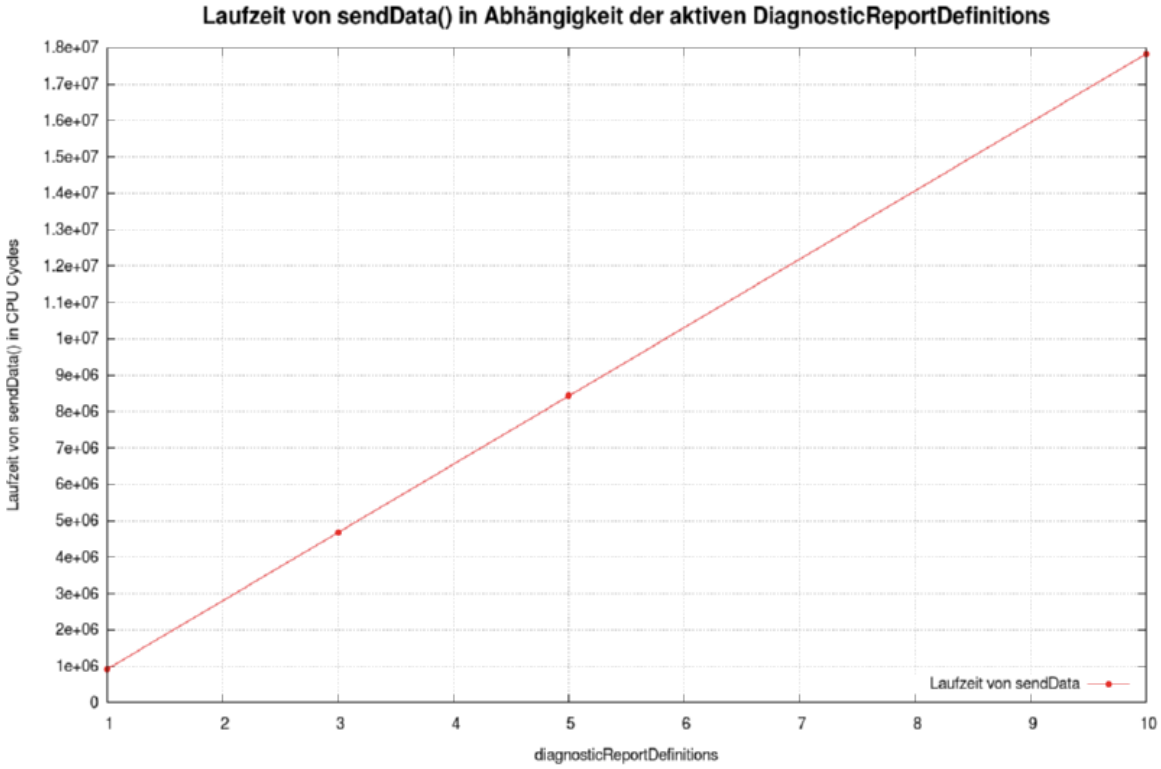
MaxDiagnosticReport Definitions	10
MaxThresholdDefinitions	10
Aktive Report Definitions & Anzahl/Art der Messages	x Reports mit je 1 IntMessage und 1 DummyMessage (Größe 8u)
Aktive Filter + Trigger pro Filter	1 Filter mit 1 Trigger (IntMessage)
Anzahl und Art von Filterable Messages (gepusht/nicht gepusht)	1 IntMessage (gepusht) und 1 VectorMessage (Größe 4000u, gepusht)
Anzahl und Art sonstiger Messages	1 DummyMessage (Größe 8u)

Messung:

x	1	3	5	10
push(int)	13800	13800	13800	13800
push(Vector)	94480	94480	94480	94480
execute	1259085	5429181	9599277	20040897
calc_delay	96720	96720	96720	96720
operator++ (Incl.)	118188	149940	181692	261072
operator++ (Self)	7252	21756	36260	72520
sendData	923733	4677933	8432133	17834013

Fazit: Wie erwartet hat die Erhöhung der aktiven DiagnosticReportDefinitions lediglich Auswirkungen auf die Laufzeit von execute. Im Gegensatz zu Test 1 wird der Laufzeitunterschied des ++ Operators nicht nur durch die Unterfunktion next() verursacht, auch die Laufzeit des ++ Operators selbst (Self-Laufzeit) steigt. Dies liegt daran, dass mit steigendem x auch die Anzahl der Aufrufe des ++ Operators steigt. Betrachtet man die Graphen 1 und 2, die die Laufzeit von operator++ und sendData gegenüber x darstellen, sieht man, dass die Zunahme der Laufzeit linear zu x verläuft, wobei die Steigung von sendData wesentlich höher ist als die des ++ Operators. Eine lineare Zunahme ist im Allgemeinen relativ unproblematisch, vor allem wenn die Steigung so gering ist wie beim ++ Operator. Die hohe Steigung von sendData ist ein Indiz, dass hier eventuell optimiert werden könnte, da sendData jedoch nur Teil des Debug Interfaces ist und später ersetzt wird wäre dies nicht sinnvoll.


Laufzeit von operator++() (Inkl.) in Abhängigkeit der aktiven DiagnosticReportDefinitions
Laufzeit von operator++ (Incl.) in CPU Cycles
270000
260000
250000
240000
230000
220000
210000
200000
190000
180000
170000
160000
150000
140000
130000
120000
110000
1
2
3
4
5
6
7
8
9
10
diagnosticReportDefinitions
Laufzeit von operator++ (Inkl.)


Laufzeit von sendData() in Abhängigkeit der aktiven DiagnosticReportDefinitions
Laufzeit von sendData() in CPU Cycles
1.8e+07
1.7e+07
1.6e+07
1.5e+07
1.4e+07
1.3e+07
1.2e+07
1.1e+07
1e+07
9e+06
8e+06
7e+06
6e+06
5e+06
4e+06
3e+06
2e+06
1e+06
0
1
2
3
4
5
6
7
8
9
10
diagnosticReportDefinitions
Laufzeit von sendData

Laufzeittest 3:
Beschreibung: Im Testprogramm wird die Anzahl an aktiven Filtern erhöht. An den Laufzeiten der Funktionen selbst sollte sich nichts ändern, die Laufzeit von sendData und somit von execute sollte jedoch zurück gehen, da durch die Filter sendData seltener Aufgerufen wird.

MaxDiagnosticReport Definitions	10
MaxThresholdDefinitions	10
Aktive Report Definitions & Anzahl/Art der Messages	10 Reports mit je 1 IntMessage und 1 DummyMessage (Größe 8u)
Aktive Filter + Trigger pro Filter	x Filter mit je 1 Trigger (IntMessage)
Anzahl und Art von Filterable Messages (gepusht/nicht gepusht)	1 IntMessage (gepusht) und 1 Vector-Message (Größe 4000u, gepusht)
Anzahl und Art sonstiger Messages	1 DummyMessage (Größe 8u)

Messung:

x	1	3	5	10
push(int)	13800	13800	13800	13800
push(Vector)	94480	94480	94480	94480
execute	20040897	18233569	16426241	11900151
calc_delay	96720	96720	96720	96720
operator++	261072	261072	261072	261072
sendData	17834013	16053153	14272293	9812373
Anzahl sendData	740	666	592	407
O(sendD.)/N(sendD.)	24100	24104	24109	24109

Fazit: Wie erwartet verändert sich lediglich die Laufzeit von sendData. Der Quotient aus der Laufzeit von sendData und der Anzahl der Aufrufe von sendData bleibt ungefähr gleich, was dafür spricht, dass der Laufzeitunterschied tatsächlich nur durch die geringere Anzahl an Aufrufen verursacht wird.

Laufzeittest 4:
Beschreibung: Im Testprogramm wird die Anzahl an Trigger (Threshold Definitionen) pro Filter erhöht. Die Laufzeitdifferenz wird von der Art der FilterableMessages, die als Trigger verwendet werden abhängen (bzw. deren isTriggered Funktion). Es wird ein geringer Laufzeitunterschied in execute erwartet, der in erster Linie vom häufigeren Aufrufen der isTriggered Funktion der verschiedenen Thresholds hervorgerufen wird.

MaxDiagnosticReport Definitions	10
MaxThresholdDefinitions	10
Aktive Report Definitions & Anzahl/Art der Messages	10 Reports mit je 1 IntMessage und 1 DummyMessage (Größe 8u)
Aktive Filter + Trigger pro Filter	x = 1: 10 Filter mit je 1 Trigger (IntMessage) x = 2: 10 Filter mit je 2 Trigger (IntMessage, VectorMessage) x = 3: 10 Filter mit 3 je Trigger (2x IntMessage, 1x VectorMessage)
Anzahl und Art von Filterable Messages (gepusht/nicht gepusht)	1 IntMessage (gepusht) und 1 VectorMessage (Größe 4000u, gepusht)
Anzahl und Art sonstiger Messages	1 DummyMessage (Größe 8u)

Messung:

x	1	2	3
push(int)	13800	13800	13800
push(Vector)	94480	94480	94480
execute	11900151	12233967	12562577
calc_delay	96720	96720	96720
operator++	261072	261072	261072
sendData	9812373	9812373	9812373
Anzahl der Trigger Abfragen	1933	3866	5799

Fazit: Ergebnis wie erwartet. Die Laufzeit steigt nur linear (siehe Grafik) und ist somit unbedenklich.


Laufzeit von execute() in Abhängigkeit der Thresholds pro Filter
Laufzeit von execute() in CPU Cycles
1.26e+07
1.25e+07
1.24e+07
1.23e+07
1.22e+07
1.21e+07
1.2e+07
1.19e+07
1
2
3
Thresholds
Laufzeit von execute

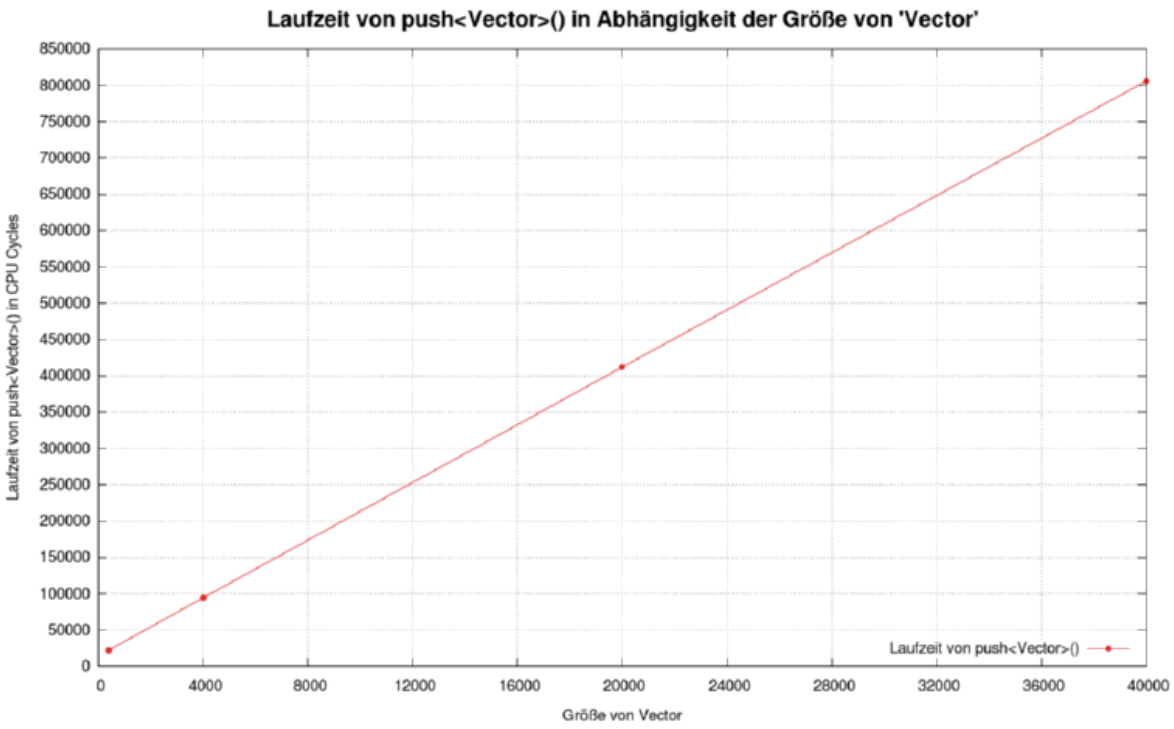
Laufzeittest 5:
Beschreibung: Im Testprogramm wird die Größe der Vector-Datenstruktur verändert. Es wird erwartet, dass sich die Laufzeit von push(Vector) ändert, da ein größerer Datenbereich kopiert werden muss. Der Laufzeitanstieg sollte dabei lediglich linear verlaufen. Da die Vector Message lediglich als Trigger dient, jedoch in keinem Report enthalten ist, bleibt die Laufzeit der execute-Funktion gleich.

MaxDiagnosticReport Definitions	10
MaxThresholdDefinitions	10
Aktive Report Definitions & Anzahl/Art der Messages	10 Reports mit je 1 IntMessage und 1 DummyMessage (Größe 8u)
Aktive Filter + Trigger pro Filter	10 Filter mit je 3 Trigger (2x IntMessage, 1x VectorMessage)
Anzahl und Art von Filterable Messages (gepusht/nicht gepusht)	1 IntMessage (gepusht) und 1 VectorMessage (Größe x, gepusht)
Anzahl und Art sonstiger Messages	1 DummyMessage (Größe 8u)

Messung:

x	400	4000	20000	40000
push(int)	13800	13800	13800	13800
push(Vector)	22480	94480	412240	805840
execute	12562577	12562577	12562577	12562577
calc_delay	96720	96720	96720	96720

Fazit: Ergebnis wie erwartet.


Laufzeit von push<Vector>() in Abhängigkeit der Größe von 'Vector'
Laufzeit von push<Vector>() in CPU Cycles
Größe von Vector
Laufzeit von push<Vector>()
850000
800000
750000
700000
650000
600000
550000
500000
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
0
4000
8000
12000
16000
20000
24000
28000
32000
36000
40000

Laufzeittest 6:
Beschreibung: Im Testprogramm wird die Anzahl der Messages verändert. Bei den FilterableMessages wird sich jeweils die Laufzeit von push erhöhen, bei den Messages, die in den Reports enthalten sind wird sich (zusätzlich) die execute Laufzeit erhöhen.

MaxDiagnosticReport Definitions	10
MaxThresholdDefinitions	10
Aktive Report Definitions & Anzahl/Art der Messages	10 Reports mit je 1 IntMessage und 1 DummyMessage (Größe 8u)
Aktive Filter + Trigger pro Filter	10 Filter mit je 3 Trigger (2x IntMessage, 1x VectorMessage)
Anzahl und Art von Filterable Messages (gepusht/nicht gepusht)	y IntMessage (gepusht) und x VectorMessage (Größe 4000u, gepusht)
Anzahl und Art sonstiger Messages	z DummyMessage (Größe 8u)

Messung:

x	1	2	3	1	1	1	1
y	1	1	1	2	3	1	1
z	1	1	1	1	1	2	3
push(int)	13800	13800	13800	27600	41400	13800	13800
push(Vec.)	94480	188960	283440	94480	94480	94480	94480
execute	12562577	12562577	12562577	12630611	12698645	12644384	12726191
calc_delay	96720	96720	96720	96720	96720	96720	96720

Fazit: Ergebnis wie erwartet.

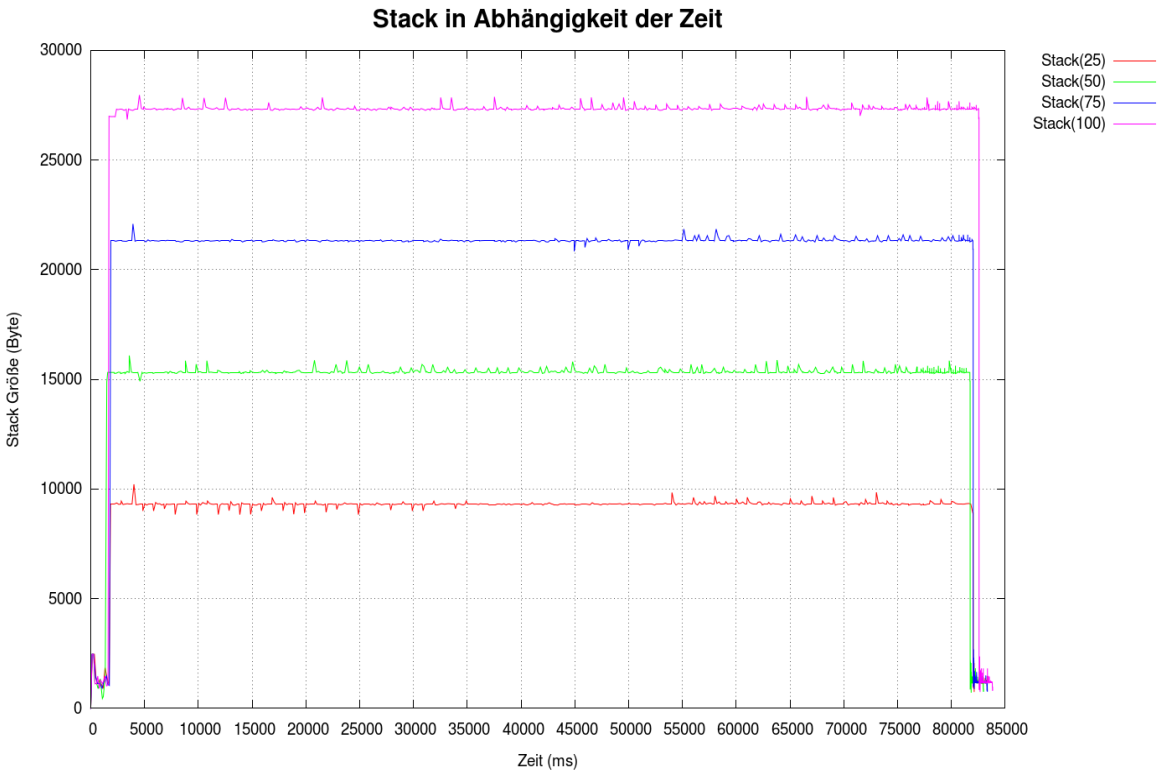
Speichertest 1:
Beschreibung: Im Testprogramm werden die Anzahl maximaler Thresholds pro Filter erhöht. Es wird erwartet, dass der Bedarf an Stack Speicher linear steigt. Heap, Text, Data und BSS sollten unverändert bleiben.

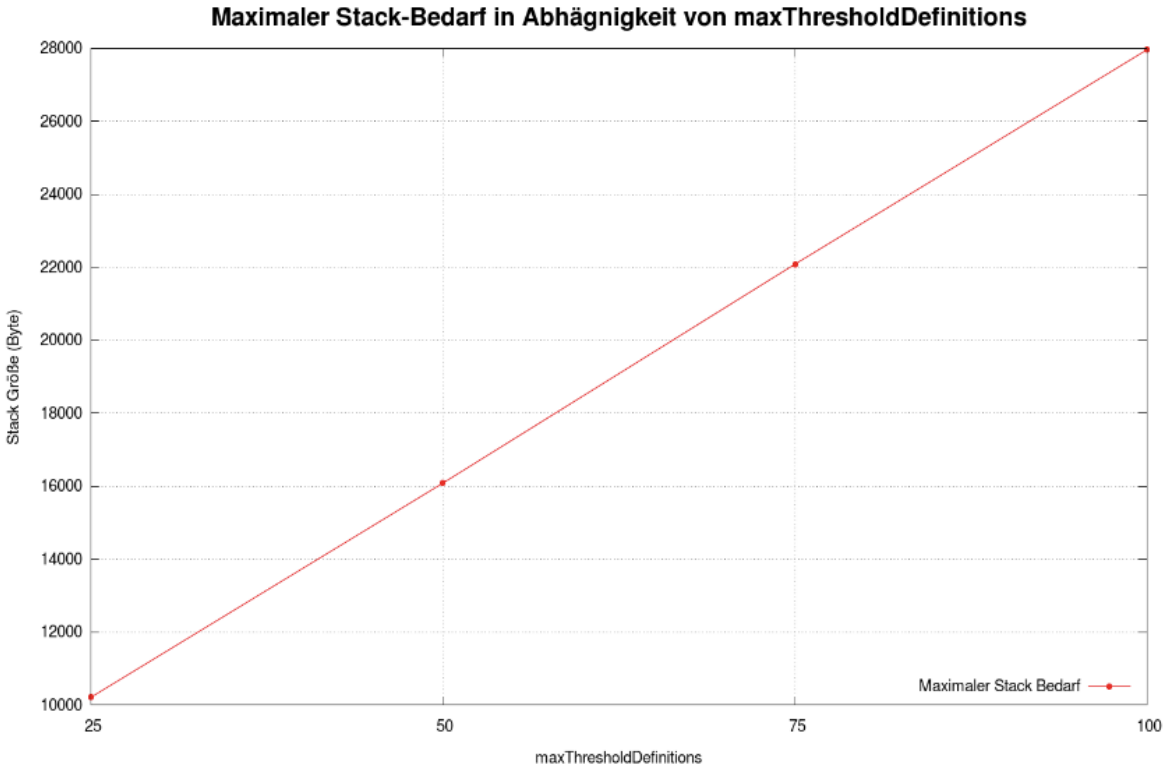
MaxDiagnosticReport Definitions	10
MaxThresholdDefinitions	x
Aktive Report Definitions & Anzahl/Art der Messages	1 Report mit 1 IntMessage und 1 DummyMessage (Größe 8u)
Aktive Filter + Trigger pro Filter	1 Filter mit 1 Trigger (IntMessage)
Anzahl und Art von Filterable Messages (gepusht/nicht gepusht)	1 IntMessage (gepusht)
Anzahl und Art sonstiger Messages	1 DummyMessage (Größe 8u)

Messung:

x	25	50	75	100
Max Stack Size	10216	16088	22088	27964
Max Heap Size	700	700	700	700
Text Size	84382	84350	84350	84350
Data Size	10356	10356	10356	10356
BSS Size	16184	16184	16184	16184

Fazit: Wie der Tabelle entnommen werden kann, treffen die Annahmen beinahe vollständig zu. Die Text Size (Größe des kompilierten Programmcodes) ist bei x=25 größer als bei den anderen Messungen. Dies ist vermutlich auf eine Compiler Optimierung oder eine Ungenauigkeit des „size“-Tools zurück zu führen. Außerdem sind die Maxima des Stack Bedarfs nicht exakt linear, was an der längeren Initialisierungsphase bei größeren Arrays liegen kann, wodurch die Snapshots des Speichers nicht immer an der selben Stelle im Programm erfolgen. Wie man an der ausgleichenden Geraden sehen kann, sind sie jedoch trotzdem in guter Näherung linear, weshalb die Annahme als bestätigt betrachtet werden kann.


Stack in Abhängigkeit der Zeit
Stack Größe (Byte)
Zeit (ms)
Stack(25)
Stack(50)
Stack(75)
Stack(100)
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000
55000
60000
65000
70000
75000
80000
85000


Maximaler Stack-Bedarf in Abhägnigkeit von maxThresholdDefinitions
Stack Größe (Byte)
maxThresholdDefinitions
Maximaler Stack Bedarf
10000
12000
14000
16000
18000
20000
22000
24000
26000
28000
25
50
75
100

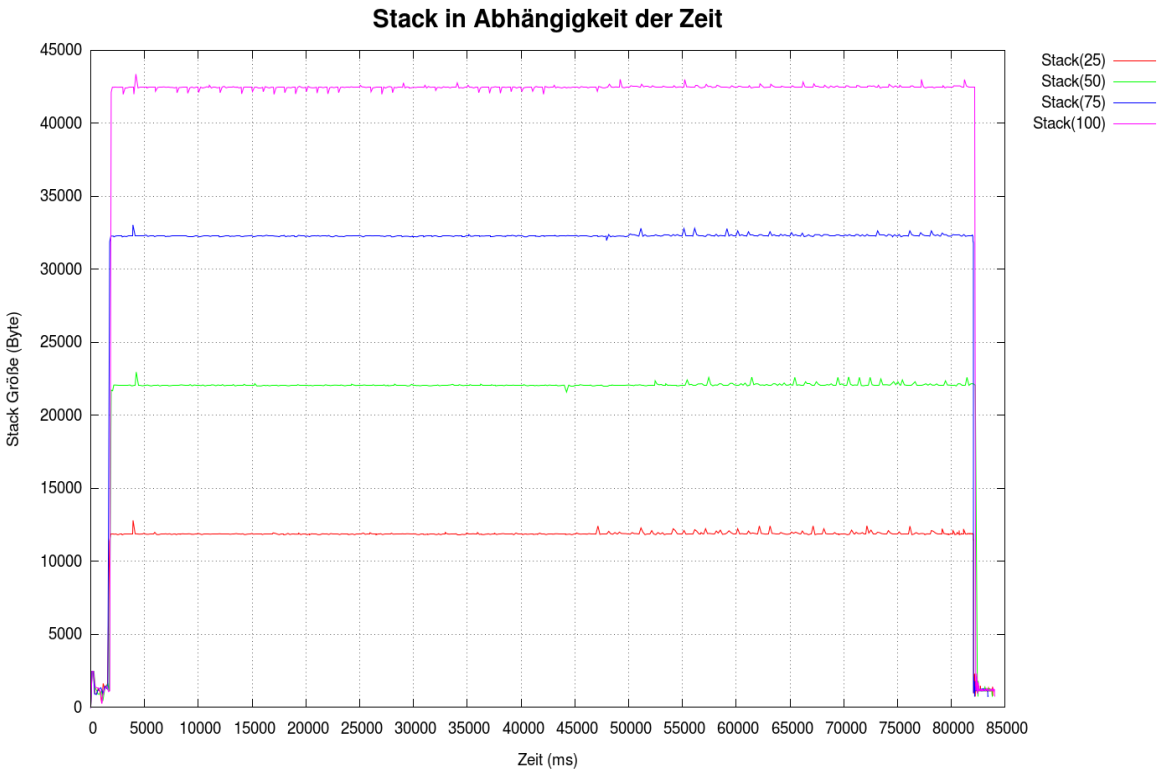
Speichertest 2:
Beschreibung: Im Testprogramm werden die Anzahl maximaler Report Definitionen erhöht. Es wird erwartet, dass der Bedarf an Stack Speicher linear steigt. Heap, Text, Data und BSS sollten unverändert bleiben.

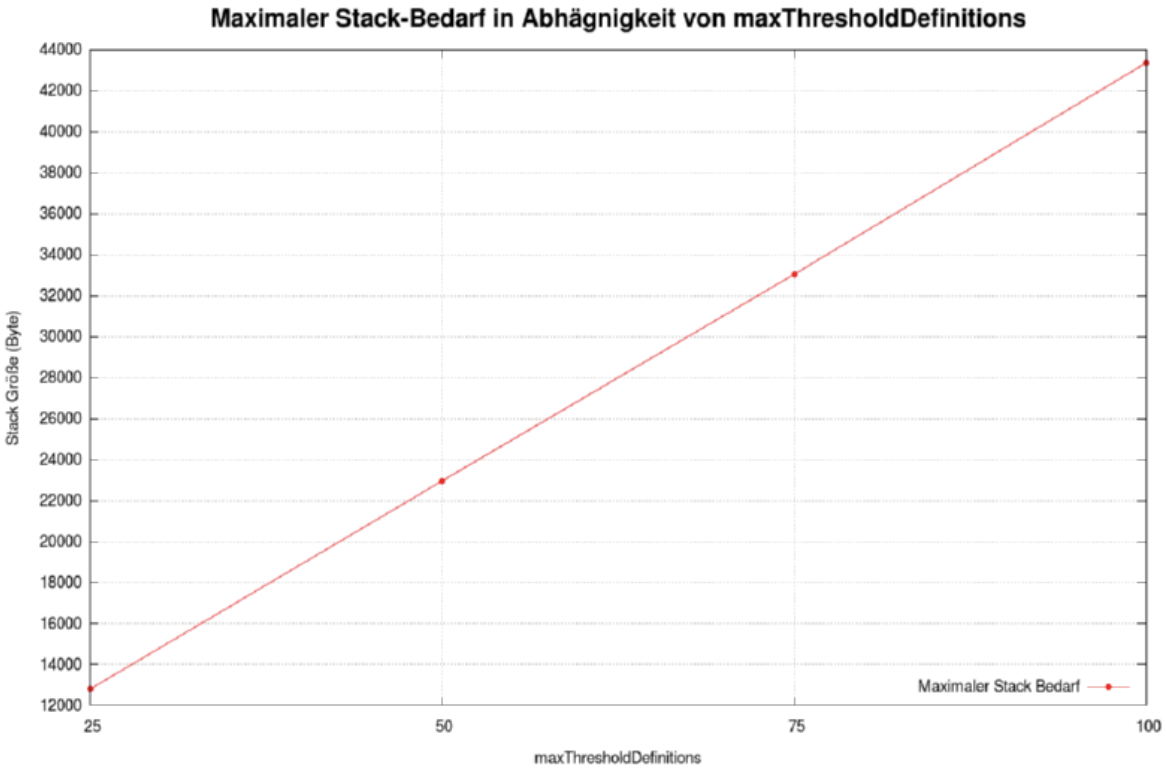
MaxDiagnosticReport Definitions	x
MaxThresholdDefinitions	10
Aktive Report Definitions & Anzahl/Art der Messages	1 Report mit 1 IntMessage und 1 DummyMessage (Größe 8u)
Aktive Filter + Trigger pro Filter	1 Filter mit 1 Trigger (IntMessage)
Anzahl und Art von Filterable Messages (gepusht/nicht gepusht)	1 IntMessage (gepusht)
Anzahl und Art sonstiger Messages	1 DummyMessage (Größe 8u)

Messung:

x	25	50	75	100
Max Stack Size	12808	22968	33048	43368
Max Heap Size	700	700	700	700
Text Size	84350	84350	84350	84350
Data Size	10356	10356	10356	10356
BSS Size	16184	16184	16184	16184

Fazit: Siehe Memory Test 1


Stack in Abhängigkeit der Zeit
Stack Größe (Byte)
Zeit (ms)
Stack(25)
Stack(50)
Stack(75)
Stack(100)
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000
55000
60000
65000
70000
75000
80000
85000


Maximaler Stack-Bedarf in Abhägnigkeit von maxThresholdDefinitions
Stack Größe (Byte)
maxThresholdDefinitions
Maximaler Stack Bedarf
12000
14000
16000
18000
20000
22000
24000
26000
28000
30000
32000
34000
36000
38000
40000
42000
44000
25
50
75
100

Speichertest 3:
Beschreibung: Im Testprogramm werden eine Reihe weiterer Variablen verändert. Der Speicherbedarf sollte nahezu unverändert bleiben.

MaxDiagnosticReport Definitions	25
MaxThresholdDefinitions	10
Aktive Report Definitions & Anzahl/Art der Messages	**10** Reports mit 1 IntMessage und 1 DummyMessage (Größe 8u)
Aktive Filter + Trigger pro Filter	**10** Filter mit **3** Trigger (3x IntMessage)
Anzahl und Art von Filterable Messages (gepusht/nicht gepusht)	1 IntMessage (gepusht)
Anzahl und Art sonstiger Messages	**2** DummyMessage (Größe 32u)

Messung:

	Test 3	Test 2 (Vergleich)
Max Stack Size	12856	12808
Max Heap Size	700	700
Text Size	84737	84350
Data Size	10356	10356
BSS Size	16184	16184

Fazit: Der Speicherbedarf hat sich wie angenommen kaum verändert. Der minimal höhere Stack Bedarf wird vermutlich durch die neu definierte DummyMessage und den dazugehörigen TaskInput verursacht. Die erhöhte Text Size kommt von den im Source Code vorgenommenen Änderungen, um die neuen Reports/Filter usw. zu definieren.

Speichertest 4:
Beschreibung: Zum Test Programm 3 wird eine FilterableMessage<Vector> hinzu gefügt (gepusht, jedoch nicht in den Reports enthalten) sowie ein Vector der gepusht werden soll. Es wird erwartet, dass sich der maximale Stack Bedarf um ungefähr 3·4000u erhöht (1x 4000 für den Vector, 2x 4000 für die Abbilder des gepushten Vectors in der FilterableMessage). Außerdem erhöht sich die Text Size etwas.

Messung:

	Test 4	Test 3 (Vergleich)
Max Stack Size	24744	12856
Max Heap Size	700	700
Text Size	87179	84737
Data Size	10356	10356
BSS Size	16184	16184

Fazit: Annahmen bestätigt.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit einschließlich aller beigefügter Materialien selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Werken entnommen sind, sind in jedem Einzelfall unter Angabe der Quelle deutlich als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden. Mir ist bekannt, dass Zuwiderhandlungen gegen diese Erklärung und bewusste Täuschungen die Benotung der Arbeit mit der Note 5.0 zur Folge haben kann.

$Unterschrift$: $Ort, Datum$: