PDP_phase_3_Maintenance_Operations.md

Contents

Introduction

This document –together with direct links to other documents– is a description of the baselined Quality Management System (QMS) that must be followed by all associates of Medical Data Networks LLC (MDN).

The goal of this QMS is to ensure that company associates and/or work products meet or exeed all of the applicable requirements for planning, manufacturing, validation, certification of Software as Medical Devices provided by MDN. This specific QMS document version is intended to address all of the requirements associated with FDA "Class II Medical Devices" realized as software services and operated by MDN.

With this particular release of the QMS, MDN is currently focused on only one "device" named **"T1Pal"**.

T1Pal provides a certified copy of the open-sourced "Nightscout" software and operates it on one or more Internet-hosted servers as a service. Such services are limited to registered subscribers. In addition, T1Pal offers bundled Technical Support services, and cryptographically-secure remote sharing features not available to other Nightcout software consumers .

While the "open-sourced" Nightscout software has proven to be the popular basis for "Do It Yourself" (DIY) projects on the Internet, the software installation and operational skills required makes the Nightscout DIY solution inaccessible to many people. In addition, the Nightscout software deployed as a DIY project may create unaddressed risks and hazards, including the risk of failures to keep the data private and secure.

Documented US FDA guidelines and requirements have set forth a specific "path" for FDA clearance of certain software services (as "medical devices").

It is the intent of this QMS to set forth all of the methods, standards, procedures, etc. as needed for MDN to realize all of its products and services as FDA "Class II Medical Devices."

The FDA "pathway" for clearance of specific software medical devices has been specifically outlined and –as applied to the present MDN– is intrepreted as follows:

1. MDN must set forth **"Indications For Use"** of its products and/or services, if any.
2. MDN must set forth the **"Intended Use"** cases for its products and/or services,
3. MDN must provide additional **special controls, validations, and quality controls** aligned with FDA requirements for labeled Intended Use of its products and/or services .

Table 1 – List of All T1Pal Intended Use Cases

Requirement ID	Intended Use Description
REQ_1010 Secondary Display	It is an intended use for T1Pal to receive data from one or more medical devices and provide a secondary display of the data.
REQ_1020 – Remote Access	It is an intended use that authorized "followers" will have remote access to a secondary display of the same data.
REQ_1030 – Technical Support	It is intended that the display of secondary data will be used to provide "Technical Support."

Note that "Technical Support" in this case is limited to providing artifacts, displays, or documentation that hasten:

1. device warranty claims,
2. guidance on replacement of devices or consumables, and
3. acts that remedy impairments to data communications.

Under no circumstances should Technical Support be interpreted as providing, correcting, or modifications of medical therapies of any kind.

In addition, with the goal of providing "special controls", "validations", and "quality controls" the following additional requirements must be met.

Table 2 – Additional T1Pal System Requirementsd

Requirement ID	Additional Requirements
REQ_2010 – Data Privacy Protections	The T1Pal is intended to be HIPPA compliant with respect to privacy and access to data.
REQ_2020 – Protection from modification of data.	The T1Pal is intended to protect against modification of data provided for secondary display.
REQ_2030 – T1Pal Labelling	The T1Pal software shall provide clear and unambiguous labelling that describes both intended use, and warnings against all other uses.

While certain services of MDN bundle open-sourced components, including Nightscout software, MDN ensures QMS controls and validation of specific Intended Use labelling, and other privacy and security features outlined above.

MDN is relying on adherence to this QMS to allow for FDA clearance of MDN products and services as a Class II software medical devices that requires no 510(k) submission, and no 513(g) request to commence marketing of the T1Pal

product.

MDN products and services are designed, marketed, and specifically limited to "secondary display" of data, Altogether, the three Intended Use cases, and the Additional System Requirements desceribed below –with the present Quality Management system– should enable MDN to meet FDA rules for clearance.

It is the responsibility of all associates, employees, and officers of the company to align their activities with this Quality Management System. The identification of "Corrective Actions" and follow-on work of company officers will be used to address non-compliance as may be needed to strengthen allignment with this Quality Management System over time.

It should be noted that this Quality Management System is captured in a "github" repository (md_qms) together with a broad range of supporting documents and templates that are expected to be used in the course of business for all Medical Data Networks LLC associates and/or employees and/or contractors. As such, the instructions are baselined in the "master" branch of this repository and subject to change only by authorized officers of the firm.

It should also be noted that in the event MDN adds products or services other than T1Pal and/or other than certain Class II Medical Devices, this QMS may be changed accordingly.

1 Control of Quality Records

This chapter establishes a standard method for completing, identifying, collecting, filing, storing, and dispositioning quality records at MDN. Quality records are maintained to provide supporting evidence of the conformity, implementation, and effective operation of the QMS.

1.1 Record Retention and Archiving.

All records and communications will be maintained in perpetuity via digital archive. Currently Medical Data Networks LLC uses these mechanisms for digital storage:

- **Google Gmail** Under the domain-specific user name

"@medicaldatanetworks.com"

all email communications are stored indefinitely, even email communications that have been deleted or archived by an individual user. Google Gmail Service is provided for every Medical Data Networks LLC associate. All company business for all matters shall use that service for all email communications.

- **Google Docs** (including Google Sheets, and related Google desktop applications) is for use in communicating informal work in progress and/or draft documents. It is not to be used for official baselined document management.
- **GitHub**, with redundant backup of the repositories of Medical Data Networks shall be stored on each and every officer's computer, and all software developers. All developers and officers of the firm shall retain their independent GitHub repository, configured to be able to synchronize with selected master copies. GitHub repositories –under control of Medical Data Networks LLC VPs and officers– is the official repository for all software, instructions, and/or Quality Management System mdn-qms documents.

1.2 Electronic Signatures

- Records requiring signature approval per above shall be digitally signed with text similar to "Approved. /s/ Name Of Approver, YYYY-MM-DD".
- Digital signatures must be affixed to documents by the electronically-authenticated user and no delegation of authority is allowed. Authentication for access to quality records for the purpose of editing or update shall be authenticated by all three means: 1) unique username/email, 2) password 3) Two-Factor Authentication via mobile device.
- Digital signatures must only be made by the actual person associated with the authentication credentials, and only over secure connections using https, TLS or SSL.

- Medical Data Networks tools that currently meet these requirements include: Gmail, Google Docs, Google Valult and GitHub.

1.3 Date and Time Marks

For most documents, dates shall be expressed in the form similar to YYYY-MM-DD. If time specification is necessary to remove ambiguity, it should be in a form similar to HH:MM:SS reflecting 24-hour clock time and local Time Zone. For example: 2016-08-15 18:43:00 EST.

1.4 Electronic Records

As generally described in the github paragraph above, all formal records at Medical Data Networks LLC are created and maintained in electronic format and maintained on a specific "github" repository.

This github repository is used to support authentication, labelling of meta data, recording of changes, and storage of the documents for each and every release of the MDN QMS .

Medical Data Networks does not maintain paper documents. If it is necessary to archive a document whose source is paper, that document is scanned or photographed and retained in the appropriate topical folder within the github management system.

1.5 Lost Documents and Disaster Recovery

Since all records and changes to records are retained in perpetuity, records should never be lost.

- If it is the case that a record cannot be found, a Corrective and Preventive Action report should be filed and an investigation conducted in accordance with chapter 3.

Source code is retained in a distributed GitHub repository that is inherently backedup by the number of cloned copies of the data.

Customer records in databases are maintained in dedicated Linux Volumes where snapshots are to be automatically taken daily.

1.6 Provision of Records to Customers

Where contractually specified, in accordance with regulatory requirements, or at the discretion of Medical Data Networks, copies of the quality records may be released to a customer, potential customer, regulatory authorities, investors, and/or lawful law enforcement personnell.

1.7 Provision of Records to Auditors or Government Agencies

Where applicable, quality records will be provided for examination during normal company operating hours at the request of an authorized quality systems auditor or government agency representative. A tool of the github repository is provided for creating a PDF export of the documents within the quality management system. However, the authoritative documents are retained within the github repository, and all printed documents and/or abstracts are uncontrolled documents. Records or reports based on records shall be made available to lawful Government Agencies, subject to the following rules:

- Internal Audit Reports, Supplier Audit Reports, and Management Review Minutes are not available for review by US FDA, according to 21 CFR 820.180(c), but may be reviewed by other regulatory agencies, as required.
- Medical Data Networks LLC officers will provide records of T1Pal operations as needed, and will make themselves available as needed to support any investigation.
- A copy of an associated log or schedule (e.g., internal audit log, management review schedule, supplier audit schedule) may be provided as proof that the activities were performed.

1.8 Responsibilities

1. The CEO and VP-level employees are responsible for overseeing and maintaining this standard operating procedure and for assuring that all employees are trained in its requirements.
2. It is the responsibility of all employees, contractors and departments at Medical Data Networks to adhere to this procedure.

2 Employee Qualification And Training

This chapter establishes the standard requirements and method for documenting employee training in Company procedures and policies, and to implement and document employee job requirements, qualifications and training programs at MDN.

These rules apply to all operators of T1Pal and/or related infrastructure services.

The following associates are also included in the scope of this document:

- An individual assigned to a job function and compensated as a "W-2" employee, full-time or part-time.
- The employees whose responsibilities include the oversight of other employees or the oversight of company matters that apply to more than one employee.
- An individual assigned to a job function, but compensated by a third-party agency or directly as a "1099" employee, full-time or part-time.
- An individual assigned a specific task or project, and providing services defined in a contract or consulting agreement.
- Any formal or informal instructions given to an employee in the form of education, which includes, but is not limited to, past experiences, formal courses, seminars, self-training or on-the-job training. Training may be either internal or external.

2.1 General Requirements

- All employees, contractors, and consultants must complete and document all required training and retraining in a timely manner.
- Training needs are identified by specific job descriptions/requirements necessary to perform particular functions. Training may be internal or external.
- As part of their training, personnel are made aware of device defects which may occur from the improper performance of their specific jobs.
- Personnel who perform verification and validation activities are made aware of defects and errors that may be encountered as part of their job functions.
- Contractors and consultants are trained on applicable MDN procedures as defined by the scope of their contract/consulting agreement.

2.1.1 Training Process Needs

1. Training needs are identified by specific job descriptions/requirements necessary to perform particular functions.
2. Personnel may receive internal training as follows, as determined by their manager:

- Self-training: training that can be initiated by the recipient by reading instructions or materials.
- Group/classroom training: more formal training conducted by a trainer.
- On-the-job training: training that takes place with a qualified trainer during the performance of an actual process. Consequences and outcomes of improper performance of employees' jobs to product quality or service will be emphasized during training.
- Training curriculum and presentation materials that are used with groups or used multiple times may be developed, reviewed and approved in accordance with the QMS document control procedures.

2.1.2 Training Records and File

- A record is maintained for each employee's training. Required training is documented and updated as needed when an employee's roles and responsibilities change due to job changes or when new responsibilities are added to a job description. Training to new or revised quality system documents are also documented.
- At a minimum, all employees are trained in the Company's quality system requirements, including regulatory requirements, the quality policy, security awareness, control of documents and records, and specific procedures and policies relevant to each job function. This training is conducted for each new employee and is repeated annually, at a minimum.
- Training activities are documented by placing a record of the training as a document stored in Google Drive.
- All personnel are responsible for maintaining and updating his/her training file.
- As part of their training, personnel are made aware of device defects which may occur from the improper performance of their specific jobs.

Personnel who perform verification and validation activities are made aware of defects and errors that may be encountered as part of their job functions.

- Contractors and consultants are trained on applicable MDN procedures as defined by the scope of their contract/consulting agreement.

2.1.3 Training Curriculum and Presentation

- Training curriculum and presentation materials that arewill be used with groups or used multiple times may be developed, reviewed and approved in accordance with QMS Document Control Procedures. For example, New Employee Orientation training materials may be given a QMS document number so that the content does not have to be prepared each time.

2.1.4 New Employee Orientation

- MDN LLC shall have sufficient personnel with the necessary education, background, training, and experience to assure that all activities required by this part are correctly performed.
- MDN LLC shall have procedures for identifying training needs and ensure that all personnel are trained to adequately perform their assigned responsibilities. Training shall be documented.
- As part of their training, personnel shall be made aware of device defects which may occur from the improper performance of their specific jobs.
- Personnel who perform verification and validation activities shall be made aware of defects and errors that may be encountered as part of their job functions.
- New Employee Orientation is completed within the first month of employment.
- A "MDN New Employee and Consultant Onboarding Checklist" is completed and stored on an internal Google Drive folder for new employees, consultants and contractors that have an @medicaldatanetworks.com email address.
- New employee orientation includes overviews of the following topics:
  - MDN development process and quality system overview.
  - MDN control of documents as described in chapter 1 "Control of Quality Records".
- Understanding and enforcement of HIPAA and other security practices at MDN.

2.1.5 Training Records

Training activities are documented by placing a record of the training as a document stored in the corporate Email system, under the user ID of the employee's supervisor.

2.2 Responsibilities

The management at MDN is responsible for:

- Maintaining personnel files, establishing employee retention policies, and establishing performance monitoring requirements.
- Developing, reviewing, and improving employee job descriptions, and training requirements.
- Completing all training and maintaining an individual training file.

3 Corrective and Preventive Action

This section establishes a standard operating procedures for corrective and preventive action.

For corrective actions, this procedure sets forth the process for determining the cause of nonconformities, initiating corrective action(s) and performing follow-up to ensure that the corrective action(s) have been effective in eliminating the cause of the nonconformity to prevent recurrence.

For preventive actions, this procedure will describe the process of identifying nonconformities and determine actions to prevent their occurrence.

This applies to all medical and research software devices designed and developed by Medical Data Networks LLC (MDN).

3.1 Sources of Nonconformities and Opportunities for Improvement

MDN may receive reports of quality issues, complaints and nonconformities, or opportunities for improvement through email, phone calls, social media, or verbal conversations.

Nonconformances may also be identified through other means such as external or internal audits, and automated monitoring tools.

Possible mechanisms for handling issues include (but are not limited to):

- The employee may generate a new Github.com repository issue or comment on an existing Github.com repository issue indicating a Bug, Feature Request, Suggestion. New Github.com repository issues may include risk analysis per QMS SOP Risk Managmeent Procedures.
- The employee may respond to the report over email, phone, or using our customer support tracking system.
- Statistical methods will be applied when necessary and as appropriate at the discretion of management. For example, emails to support@t1pal.com may be analyzed for patterns of recurring customer complaints. When performed, such analysis will be documented and stored in Google Drive.
- Based upon the analysis of these issues, it is determined whether a corrective action or preventive action is required.

At the discretion of MDN management or employees, non-conformities that involve severe operational issues, security issues or invalid data may also be detailed in a root cause analysis document.

3.2 Verification of Effectiveness

- MDN verifies that the corrective or preventive action does not adversely affect the ability to meet applicable regulatory requirements or the safety and performance of the medical device.
- MDN establishes a date to determine the effectiveness of the corrective or preventive action. Effectiveness verification shall include objective evidence to demonstrate that the corrective or preventive action is effective.

3.3 Dissemination of Information

- All information will be disseminated to appropriate MDN employees using email, and/or a Github repository owned by MDN.
- Management will review all relevant information. Decisions about corrective and preventive action to be taken in current or future activities will be documented as Github.com repository issues. This information is also presented at Management Operations Review meetings.

3.4 Documentation

All documentation regarding quality issues and corrective and preventive action are permanently store in the QMS “Post Mortem” document store folder.

4 Risk and Hazard Management

The purpose of this chapter is to define the risk management process used by MDN to document and maintain an ongoing process for identifying hazards associated with MDN medical devices, estimating and evaluating the associated risks, controlling these risks, and monitoring the effectiveness of the controls.

It should be generally understood that ALL references to manufactured "device(s)" mentioned in the sections below are applicable to T1Pal services delivered by one or more compute environments operated by MDN.

MDN does not manufacture tangible devices, but rather is leveraging the FDA approach to define "software applications" as manufactured devices, subject to rules and regulations of devices, where possible.

This Quality Management System is designed to meet or exceed the applicable ISO standards for Quality Management Systems, and Risk Management standards found in the following references.

The plans and practices outlined in these references are believed to be entirely contained in the specific definitions, Plans, and Risk Assessment Activities described below the reference section. If and/or when there is some difference in the references compared to to the QMS details following the references, the QMS documentation is authoritative and is used.

1. ISO 14971:2019 ISO Medical Devices – Application of risk management to medical devices.
2. ISO 13485:2016 ISO Medical Devices – Quality Management Systems – Requirements for regulatory purposes.
3. 21 CFR Part 820, US FDA Quality System Regulation – FDA Code of Federal Regulations Title 21 – Part 820 Quality System Regulation

4.1 Definitions

- Harm - Physical injury or damage to the health of people, or damage to property or the environment
- Hazard - Potential source of harm
- Hazardous Situation - Circumstance in which people, property, or the environment are exposed to one or more hazard(s)
- Life-cycle – All phases in the life of a medical device, from the initial conception to final decommissioning and disposal
- Post-production – Part of the life-cycle of the product after the design has been completed and medical device has been manufactured
- Residual Risk – Risk remaining after risk control measures have been taken
- Risk – Combination of the probability of occurrence of harm and the severity of that harm
- Risk Analysis – Systematic use of available information to identify hazards and to estimate the risk

- Risk Assessment – Overall process comprising a risk analysis and a risk evaluation
- Risk Control – Process in which decisions are made and measures implemented by which risks are reduced to, or maintained within, specified levels
- Risk Estimation – Process used to assign values to the probability of occurrence of harm and the severity of that harm
- Risk Evaluation – Process of comparing the estimated risk against given risk criteria to determine the acceptability of the risk
- Risk Management – Systematic application of management policies, procedures and practices to the tasks of analyzing, evaluating, controlling and monitoring risk
- Risk Management File (RMF) – Set of records and other documents that are produced by risk management
- Safety – Freedom from unacceptable risk
- Senior Management – CEO and other senior executives
- Severity – Measure of the possible consequences of a hazard
- Use error – Act or omission of an act that results in a different medical device !response than intended by the manufacturer or expected by the user
- Verification – Confirmation, through the provision of objective evidence, that specified requirements have been fulfilled

4.2 Baselined Risks

It is assumed that all tasks have "baseline risks". Baseline risks are documented on an MS Excel Spreadsheet stored in the software github repository.

New Baseline Risks are added iteratively. If a task or process does NOT indicate additional risks, then only the Baseline Risks are assumed.

Risk analysis occurs regularly and continuously as a part of all processes. If a task is determined to have risk(s), then those risk(s) are documented.

Supporting the uniform analysis of risk, the following tables provide the working definition of "Severity" and "Harm".

4.2.0.1 Severity Definitions

Rating	Term	Description	Short example
5	Catastrophic	Results in patient death.	Death due to hypoglycemia.
4	Critical	Results in permanent impairment or life-threatening injury.	Hypoglycemic coma.

Rating	Term	Description	Short example
3	Serious	Results in injury or impairment requiring professional medical intervention.	Hypoglycemic seizure or diabetic ketoacidosis requiring hospitalization. Serious injury due to hypoglycemia-induced fainting.
2	Minor	Results in temporary injury or impairment not requiring professional medical intervention.	Confusion or disorientation due to minor hypoglycemia.
1	Negligible	Inconvenience or temporary discomfort.	Feeling a little low, quickly recovering.
0	None	A bug or issue that has no chance of causing harm.	Minor user interface issue that will not cause misinterpretation of data. Bug found and fixed prior to delivery to production

4.2.0.2 Definition of "Probability of Harm" Definitions

Rating	Term	Probability (P) of Occurrence of the Harm (not the bug)	Description for Clarity
5	Frequent	P 0.1	Likely to occur 1 in 10 times or more often.

Rating	Term	Probability (P) of Occurrence of the Harm (not the bug)	Description for Clarity
4	Probable	.01 P < 0.1	Likely to occur between (1 in 10) and (1 in 100) times.
3	Occasional	.0001 P < 0.01	Likely to occur between (1 in 100) and (1 in 10,000) times.
2	Remote	.000001 P < 0.0001	Likely to occur between (1 in 10,000) and (1 in 1,000,000) times.
1	Improbable	P 0.000001	Likely to occur less frequently than 1 in a million times.

Probability of a risk is determined by estimating the likelihood that the risk will manifest as a harm for any given use of MDN software. “Use” means an instance of using a MDN software for its intended use.

4.3 Risk Rating

A Risk Rating is determined by multiplying the occurrence rating by the severity rating to obtain a result ranging from 1-25. Decisions about risk acceptability are based on the following table:

Severity	None	Negligible	Minor	Serious	Critical	Catastrophic
Probability						
Frequent	0	6	13	17	21	25
Probable	0	5	12	16	20	24
Occasional	0	4	8	15	19	23
Remote	0	2	7	14	18	22
Improbable	0	1	3	9	10	11

4.4 Risk Management Procedures

1. All software and medical devices have risks. MDN's risk management activities reduce risk.
2. Unless otherwise documented, all MDN software is assumed to have Baseline Risks, as defined below.
3. All risks, including Baseline Risks, are analyzed, estimated, evaluated and documented as described below.
4. Risk Management activities, including review of the Risk Management process, are iterative and continuous. There are no point-in-time "Risk Acceptance" gates.
5. MDN's risk management process follows the process identified in ISO 14971 Recognized Consensus Standard.
6. MDN's CEO ensures that there are adequate resources for the risk management process and that these personnel are qualified for risk management with the knowledge and experience appropriate to the tasks assigned to them.

4.5 Risk Management Plan and File

MDN has designed its own Risk Management Plan based on analysis of existing state of the art for software and medical device. We base our model on these tenets:

1. **Reduction** of risk through software iteration over time.

Processes that inhibit iteration cycles add risk. MDN uses agile design and development methodology, including verification and validation processes. The more quickly we can iterate on our software, the more quickly we can converge on high quality software that meets the user's needs while maintaining (or improving) software safety and efficacy.

2. **Comparison** of risks and mitigations, including comparison to baseline risk of living with diabetes.

All software and medical devices contain risks and flaws, both known and unknown. Diabetes is also an inherently risky disease. Existence of a residual risk in MDN's software must be compared against the risk of unavailability of MDN software or other diabetes management software for a person living with diabetes.

MDN establishes a risk management plan for each medical device. This plan includes the scope of the risk management activities, assignment of responsibilities and authorities, criteria for risk acceptability, verification activities and activities related to the collection and review of production and post-production information.

This risk management plan, risk evaluation, implementation and verification of risk control measures, and the assessment of acceptability of any residual risks

comprise the Risk Management File. The Risk Management File is an index with pointers to the required documentation.

4.6 Risk Analysis

1. For each medical device, MDN documents the intended use and reasonably foreseeable misuse, as well as documenting qualitative and quantitative characteristics that could affect the safety of the medical device. A form "Device Characteristics" is used to document the results of this review.
2. Hazardous situation(s) are recorded after conducting an assessment of reasonably foreseeable sequences or combinations of events that can result in a hazardous situation.
3. The associated risk for each hazardous situation is estimated using available information or data. Risk estimation is based on assigning Severity and Probability to a given hazard. Severity and Probability are defined below.
4. For each identified risk, the qustion: "What is the severity of this risk? is documented.

4.7 Risk Acceptability

1. Risk values 1, 2 and 3: Further risk reduction will be assessed. Once this risk has been reduced as far as possible, this risk is acceptable for users to continue using MDN software with these residual risks. For known risks that may have potential fixes or mitigations, the potential fix or mitigation should be documented so that it can be considered relative to future work during future prioritization.
2. Risk value 4 through 11: Risk reductions must be considered and a mitigation plan must be documented and put in place. Once these risks have been reduced as far as possible, these risks are considered acceptable.
3. Risk values 12 through 25: These risks must be reduced. The work must be prioritized and addressed ahead of other work on the same feature area. A mitigation plan must be documented and put in place. A communication plan to users must be documented and implemented. Prior to product release, risk must be reduced to an acceptable level or a Risk/Benefit analysis must be performed prior to product release for human use. If the medical benefits outweigh the overall residual risk, then the overall residual risk can be judged acceptable.

4.8 Risk Evaluation

1. For each identified hazardous situation, MDN will decide whether the risk can be further reduced. Any requirement for risk reduction, or if none is required, is recorded in the risk analysis.

2. All risks shall be reduced as far as possible.
3. Risks will generally be categorized as Acceptable or Unacceptable. Any risks determined unacceptable require risk reduction prior to going to market.

4.9 Risk Control

1. When risk reduction is required, risk control activities as described below will be performed.
2. In the event that risk reduction is necessary, risk control measures will be identified and documented. Risk control measures are applied in the following priority:
    - Safety by design (inherent) - eliminates hazard or hazardous situation via design feature
    - Protective measures - prevent or reduce likelihood of occurrence of hazard or hazardous situation
    - Information for safety - warnings, precautions, and/or information provided regarding hazard or hazardous situation
3. The risk control measures that are identified are verified and the effectiveness of the risk control measures, which may include validation activities, are documented accordingly.
4. Upon implementing the risk control measures, the residual risks are then evaluated according to the criteria established above. Generally, if the residual risk has a risk rating 11, then it is determined to be acceptable.
5. If residual risks are still determined to be unacceptable, additional risk controls will be identified and implemented.
6. For residual risks that have been reduced as low as possible, top management will decide which residual risks to disclose and what information for safety is necessary to include in the Instructions for Use in order to disclose those residual risks.
7. Following the residual risk evaluation, an assessment is made as to whether the medical benefits of the device outweigh the residual risks. Clinical evaluation of the product and any other relevant published clinical literature may be considered in making the determination. If the evaluation demonstrates that the medical benefits of the device do not outweigh the residual risks, the device is not released to production.
8. The risk control measures that have been implemented are also evaluated to determine if they have introduced any new hazards or hazardous situations and whether the estimated risks for previously identified hazardous situations are affected.

9. Once all risk control measures have been implemented, a review is performed to determine whether the risks from all identified hazardous situations have been considered.

4.10 Evaluation of Overall Residual Risk Acceptability

1. After all the risk control measures have been implemented and verified, it will be determined whether the overall residual risk posed by the medical device has been reduced as low as possible using the criteria defined in the Risk Management Plan.
2. If the overall residual risk is not judged to be reduced as far as possible using the criteria established in the Risk Management Plan, then further data and literature may be gathered and reviewed to determine if the medical benefits of the intended use outweigh the overall residual risk. If the evidence supports that the benefits outweigh the overall residual risks, the the overall residual risk can be judged to be as low as possible. Otherwise, the overall residual risk remains unacceptable.
3. Risk Management Report: Prior to release, MDN reviews the risk management process to ensure that it has been appropriately implemented and that the overall residual risk is accepted.
4. These results will be documented in the Risk Management Report and included in the Risk Management File.
5. Production and Post-production Information

On an iterative, ongoing basis, complaints, bug reports and other user feedback are compiled and evaluated per chapter 3 “Corrective and Preventive Action” and chapter 5 “Complaint Handling”

Notes on “As Far As Possible”

The term and understanding of the phrase for reducing risk “As Far As Possible” is hereby understood to be a holistic definition that balances unit cost, business goals, the incremental benefits and impact for improving risk. What is “possible” is defined in the context of a specific and contemporary “willingess to pay” for the realization of the “possible.” If there is no one willing to pay for the possible then, by definition, it is not possible.

5 Complaint Handling

This procedure outlines how Medical Data Networks LLC (MDN) receives, reviews, and evaluates complaints associated with T1Pal operations. It is intended to be compliant with FDA Title 21 Part 820 Section 820.198 Complaint files.

This procedure is used by Customer Operations personnel, who have the necessary education, background, training and experience to receive and document complaints for all of MDN's products.

5.1 Definitions

5.1.1 "Complaint"

Any written, electronic, or oral communication that alleges deficiencies related to the identity, quality, durability, reliability, safety, effectiveness, or performance of an MDN product or service after it is released for commercial distribution.

5.1.2 "Inquiry"

A user question regarding the use of a MDN product that does not meet the definition of a complaint (e.g., a question about how a feature works, or how to update apps).

If repeated inquiries are received from different users regarding the same topic, corrective action may be needed because repeated inquiries of the same nature may indicate a user/use problem.

5.1.3 "Malfunction"

A Malfunction is the failure of a device to meet any of its performance specifications or otherwise to perform **as intended**.

Performance specifications include all claims made **in the labeling for the device**. The intended performance of a device refers to the objective **intent of the person legally responsible for the labeling** of the device.

5.1.3.1 "Serious Injury Definition" An injury that:

- Is life threatening,
- Results in permanent impairment of a body function or permanent damage to body structure, or
- Necessitates medical or surgical intervention by a healthcare professional to:
  - preclude permanent impairment of a body function or permanent damage to body structure or

– relieve unanticipated temporary impairment of a body function or unanticipated temporary damage to body structure.

Temporary impairment of a body function or temporary damage to body structure is unanticipated if reference to such impairment or damage is not made in the labeling for the device or, if such reference is made in the labeling for the device, the manufacturer or importer of the device determines that such impairment or damage has occurred or is occurring more frequently or with greater severity than is stated in the labeling for the device or, if there is not any pertinent statement in the labeling, than is usual for the device.

Properly labelled, a product/service offering a secondary display which further, excludes use cases covering the calculation or delivery of therapies, would be unexpected to fail in such a way to create a reportable event.

5.2 User Communication

Any contact (phone, email, support ticket, website inquiry, in person) from a user regarding an MDC product or service.

5.3 Complaint Handling Policy

MDN may receive Complaints through any form of User Communication. However, all Complaints will be submitted to, or forwarded to MDN's Support team within 1 business day. This is accomplished when a support ticket is created on the technical support web page.

In addition, this particular ticketing system facilitates the collection of additional detail by linked email exchanges as needed to fully explain the complaint and/or any conditions surrounding the complaint. Therefore, complaints may also be received when an email is sent to support@t1pal.com and/or support@t1pa.helpy.com.

All Complaints will be evaluated by an MDC Vice President or higher rank employee.

Complaints that meet the definition of a Reportable Event will be clearly tagged within the ticketing system and added to (or used to update) a Reportable Event Spreadsheet maintained by MDN Quality VP.

All Reportable Events will be promptly investigated by a MDN quality VP. Risk analysis, mitigation and control will be performed per chapter 4 "Risk and Hazard Management".

Complaints that are not deemed to be Reportable Events will be evaluated regularly as part of a routine review.

Reportable Events that result in Serious Injury will be reported to the appropriate regulatory authority.

5.4 Requirements for Maintaining Complaint Files.

1. MDN shall maintain complaint files. Each manufacturer shall establish and maintain procedures for receiving, reviewing, and evaluating complaints by a formally designated unit. Such procedures shall ensure that:
   - All complaints are processed in a uniform and timely manner;
   - Oral complaints are documented upon receipt; and
   - Complaints are evaluated to determine whether the complaint represents an event which is required to be reported to FDA under part 803 of this chapter, Medical Device Reporting.
2. Each manufacturer shall review and evaluate all complaints to determine whether an investigation is necessary. When no investigation is made, the manufacturer shall maintain a record that includes the reason no investigation was made and the name of the individual responsible for the decision not to investigate.
3. Any complaint involving the possible failure of a device, labeling, or packaging to meet any of its specifications shall be reviewed, evaluated, and investigated, unless such investigation has already been performed for a similar complaint and another investigation is not necessary.
4. Any complaint that represents an event which must be reported to FDA under part 803 of this chapter shall be promptly reviewed, evaluated, and investigated by a designated individual(s) and shall be maintained in a separate portion of the complaint files or otherwise clearly identified. In addition to the information required by §820.198(e), records of investigation under this paragraph shall include a determination of:
   - Whether the device failed to meet specifications;
   - Whether the device was being used for treatment or diagnosis; and
   - The relationship, if any, of the device to the reported incident or adverse event.
5. When an investigation is made under this section, a record of the investigation shall be maintained by the formally designated unit identified in paragraph (a) of this section. The record of investigation shall include:
   - The name of the device;
   - The date the complaint was received;
   - Any unique device identifier (UDI) or universal product code (UPC), and any other device identification(s) and control number(s) used;
   - The name, address, and phone number of the complainant;
   - The nature and details of the complaint;

- The dates and results of the investigation;
- Any corrective action taken; and
- Any reply to the complainant.

6. When the manufacturer's formally designated complaint unit is located at a site separate from the manufacturing establishment, the investigated complaint(s) and the record(s) of investigation shall be reasonably accessible to the manufacturing establishment.
7. If a manufacturer's formally designated complaint unit is located outside of the United States, records required by this section shall be reasonably accessible in the United States at either:
   - A location in the United States where the manufacturer's records are regularly kept; or
   - The location of the initial distributor.

5.5 Complaint Handling Flowchart

A flowchart that describes the complaint handling procedure is available in the training materials.

6 Medical Device Reporting

This standard operating procedure establishes MDN's Medical Device Reporting (MDR) process and responsibilities under 21 CFR 803 Medical Device Reporting.

It is important to note that within the bounds of this QMS, the T1Pal service (provided by a collection of operating servers on the internet) is defined as a "device" subject to all of the definitions, protections, certifications, and rules applicable to Medical Devices, as defined by the US FDA. We therefore take all uses of the term "device" in FDA documents to be inclusive of services such as T1Pal that is operated and maintained by Medical Data Networks LLC for the benefit of its lawful subscribers.

6.1 Definition of a "Reportable Event"

Unless otherwise noted, all terms take on the meaning as defined in chapter 7 "Complaint Handling" and 21 CFR 803 Medical Device Reporting.

A "reportable event" means:

1. An event that user facilities become aware of that reasonably suggests that a device has or may have caused or contributed to **a death or serious injury** or
2. An event that manufacturers or importers become aware of that reasonably suggests that one of their marketed devices:

(i) May have caused or contributed to a death or serious injury, or

(ii) Has malfunctioned and that the device or a similar device marketed by the manufacturer or importer would be **likely to cause or contribute to a death or serious injury** if the malfunction were to recur.

Note that as MDN provides only one product "T1Pal" that is defined as a "secondary display" – not labeled for theraputic use, and not labeled for any use when a primary display is not available or otherwise inoperable, T1Pal software cannot –by definition– contribute to a death or serious injury on its own.

6.2 Summary of Reporting Policy

1. If MDN becomes aware of a Reportable Event, MDN will file an MDR as follows:

i. within 5 days of becoming aware that a Reportable Event requires remedial action to prevent an unreasonable risk of substantial harm to the public health;
ii. within 5 days of becoming aware of a reportable event for which the FDA has made a written request;

iii. within 30 days after becoming aware of a reportable death, serious injury, or reportable malfunction.

2. If it becomes necessary for MDN to file an MDR, it will be done so electronically following instructions from here: https://www.fda.gov/ForIndustry/FDAeSubmitter/ucm108165.htm

3. Documentation for MDRs will be maintained according to chapter 1 of this QMS "Control of Quality Records"

6.3 Detailed Requirements

Devices that "may have caused or contributed" to a death or serious injury; or a malfunction that would be likely to cause or contribute to a death or serious injury must be reported.

Medical Data Networks LLC (MDN) must submit death, serious injury, and malfunction reports within 30 days after they become aware of a reportable event. The information can come from any source.

Reasonably Known:

Firms must provide all information that is reasonably known to them. FDA considers the following to meet this standard, i.e., any information: that can be obtained by contacting a user facility, distributor, and/or other initial reporter, in the manufacturer's possession, that can be obtained by analysis, testing, or other evaluation of the device.

Information required to be reported:

The form FDA 3500A is the primary reporting form for death, serious injury and malfunction events.

With the exception of drug or biologic related items, all the fields must be completed or have an entry (NA, NI, or UNK) indicating why the information could not be obtained.

Missing Information:

Manufacturers are responsible for obtaining and providing FDA with any information that is missing from reports that are received from user facilities, distributors, and other initial reporters.

If a firm cannot provide complete information, it must provide a statement explaining why such information was incomplete and the steps taken to obtain the information. Any required information not available at the time of the report, obtained at a later date, must be forwarded to FDA in a supplemental report within one month of receipt.

Investigation:

Manufacturers are responsible for investigating and evaluating the cause of each event. These investigations must follow the requirements in 21 CFR 820.198 and provide the information required on form FDA 3500A, Block H.6, H.7, and H.9.

Five-Day Reports - 803.53:

Manufacturers must submit a five-day report on form FDA 3500A within five days under the following two conditions: * 1. They become aware that an MDR reportable event, from any source, requires remedial action to prevent an **unreasonable risk** of **substantial harm** to the public health. OR * 2. They receive an FDA written request for the submission of five-day reports.

Baseline Reports - 803.55:

Manufacturers are required to submit a baseline report on FDA 3417 form when the device model is first reported under 803.50. Baseline Reports must be updated annually (if information changes) on the firm's scheduled registration date, as required by Part 807.21.

Supplemental Reports - 803.56: Manufacturers are required to submit, within one month after receipt any required information regarding deaths, serious injuries, and malfunctions that was not available to them when the initial report was submitted.

6.4 General Medical Device Reporting Guidance

This document provides general guidance regarding the reporting of adverse events required by the Medical Device Reporting (MDR) Regulation.

6.4.1 The Obsolete Per-se Rule

The submission of an event by a health care professional does **not** require the manufacturer to report the event based solely on the statements of a health care professional. The event must meet the reporting criteria in MDR to qualify as a reportable event.

6.4.2 Reporting Time Frame

Firms now have up to 30 CALENDAR days after they become aware of a device related death, serious injury or malfunction before they are required to submit a report to FDA.

6.4.3 Five-Day Reports

Five-day reports are required in two circumstances.

First, they are required if a manufacturer becomes aware that a reportable event, from any source of information, necessitates remedial action to prevent **an unreasonable risk** of **substantial harm** to the public health.

Second, five-day reports are required when a manufacturer becomes **aware of an MDR reportable event** for which FDA has requested a five-day report.

6.4.4 Non-Reportable Events

Firms must submit MDR reports when the reported information reasonably **suggests an association** between one of its devices and a reportable **death, serious injury or malfunction**.

Under some circumstances, an adverse event may appear to trigger the requirement of submission of an MDR, but because information reveals the device did not cause or contribute to the death or serious injury, no MDR is required. Thus as described below, a manufacturer will have to investigate the event in order to know if it should be reported.

A firm is required to submit an MDR report when it becomes aware of information reasonably suggesting that an event meets the criteria for reporting a Death, Serious Injury, or Malfunction. For example, a hospital informs a manufacturer that its device has failed and, as a result, a patient died. At this point, the firm has become aware of information that reasonably suggests they are in receipt of a reportable MDR event.

Next, the firm must investigate the report to determine its cause.

Both the QS Regulation and MDR require investigation of complaints.

During its investigation a firm may become aware of information that changes the initial report's conclusions. For example, the firm may find that its device was not involved in the death and could not have caused or contributed to the death. In these instances the firm would document the information that changes the association between its device and the death. No report would be required if the death or other facts turn out to be incorrect. But, if the firm becomes aware of the identity of the device/firm that was associated with the death, the firm is responsible for forwarding the information to the FDA.

However, if the firm's investigation does not change the alleged association between the device and the death, the event must be submitted as an MDR report.

In addition, if the firm's investigation produces information that would cause a person who is qualified to make a medical judgment to reach a reasonable conclusion that the device did not cause or contribute to a reportable MDR event - no report is required. Translation - if a firm decides NOT to report an apparent device-related death, serious injury or malfunction - this decision must be made by a person that the regulation recognizes as qualified to make a medical judgment, i.e., a physician, nurse, risk manager, or biomedical engineer.

Using the example from above, if the firm's investigation yields an autopsy finding that the patient died from cancer – not the device - the firm could decide NOT to report as long as the decision is consistent with the regulation: There is documented information that changes the association between the death and

the device, the decision is made by a person who is qualified to make a medical judgement, and the conclusion reached by the person in item two is reasonable.

PLEASE NOTE THE FOLLOWING:

Firms ARE NOT required to have every MDR report reviewed by a person qualified to make a medical judgement and/or a person with a medical degree or training.

Individuals who are not qualified to make a medical judgement can review MDR reports and make decisions on the basis of facts but they cannot make decisions NOT to report MDR events that require medical judgement.

In lieu of in-house or on-site qualified medical personnel or individuals qualified to make a medical judgement the firm may use consultants.

When reviewing a non-reportable event validate and document the credentials of the individual making these decisions as well as the decision not to report the event.

6.4.5 Investigation

Firms are required to investigate EVERY device related death, serious injury and malfunction in accordance with QS regulation, 820.198.

Failure to comply with this provision is a violation of BOTH the QS regulation and MDR.

Manufacturers are also required to VERIFY information on each form FDA 3500A as well as make a good faith effort to obtain information that is missing/not provided by the reporter. If the firm cannot obtain the missing information, the MDR complaint files shall contain an explanation of why the information could not be obtained as well as documentation of the firm's efforts to obtain the missing information.

6.4.6 Reasonably Known Information

FDA considers information that can be obtained by contacting the reporter to be in the possession of a firm, and considers information that can be obtained by analysis, testing, or other evaluation of a device to be information that a firm is expected to REASONABLY know, obtain and report.

6.4.7 Reasonably Known Good Faith Effort

A firm must demonstrate that it exercised "good faith" in any failed attempts to obtain required data that is missing, incorrect, or that FDA considers to be reasonably known. While the concept of good faith is generally considered to be equivalent to "due diligence", CDRH has not developed a standard. However, the firm's procedures for obtaining missing information should appear under the "Internal Systems" section of its written MDR procedures. In addition, the

Center believes that the parameters of good faith effort must, at a minimum, comport with the level of risk/nature of the device associated with the event being investigated.

6.4.8 Serious Injury

The interpretation of what constitutes a serious injury can be subjective and complicate the enforcement of MDR. The "**unanticipated temporary impairment**" part of the former serious injury definition has been rescinded, thus alleviating a source of subjectivity. In addition, the requirements that intervention be "immediate" and the concept of "probability" have also been removed from the serious injury definition.

The current MDR regulation states that **a serious injury is an "injury or illness**." This literally means that **there has to be an injury that is life-threatening, results in permanent impairment/damage, or necessitates medical/surgical intervention to preclude permanent impairment/damage** in order for an event to be reportable as a serious injury.

If there is no injury attributable to the device, then there is no serious injury report, however, the event may qualify as an **MDR reportable malfunction** depending upon the circumstances.

The Center may decide to clarify the definition of serious injury. These categories will be provided to the field and the industry through MDR guidance documents and/or letters, as necessary.

6.4.9 Malfunctions

Malfunction reporting decisions have been the subject of concern by both industry and the FDA. Basically, a malfunction is **an event that is likely to cause or contribute to either a death or serious injury**, but some circumstance prevented the injury or death from occurring. These events are very important since they represent "potential" deaths or serious injuries and provide the Agency with the opportunity to be proactive in reducing risks.

Not all malfunctions, however, are MDR reportable events.

If a malfunction is not reportable as an MDR, it may be a complaint and thus subject to the QS complaint handling requirements. Determining if an event is a reportable malfunction involves answering a number of questions including:

- Is the event device-related?
- Has the device failed to perform its intended function or meet its performance specifications?
- Is this failure likely to cause or contribute to a death or serious injury if the event were to happen again?

There is a presumption in the MDR regulations that if the event happened once it can happen again. The determination of whether to submit a report should

be based on the potential outcome. For example, if this malfunction were to occur, how would it affect the patient? If the answer is "the malfunction is likely to cause or contribute to death or serious injury" then the event is reportable. The preamble to the MDR regulations (Federal Register: December 11, 1995, Volume 60, Number 237, pages 63577-63607) offers the following guidance for determining circumstances in which malfunctions should be reported

- The chance of a death or serious injury occurring as a result of the recurrence of the malfunction is not remote;
- The consequences of the malfunction affect the device in a catastrophic manner that may lead to a death or serious injury;
- The malfunction results in the failure of the device to perform its intended essential function and compromises the device's therapeutic, monitoring or diagnostic effectiveness, which could cause or contribute to a death or serious injury.

NOTE: The essential function of a device refers, not only to the device's labeled use, but for any use widely prescribed within the practice of medicine.

The malfunction involves a long-term implant or a device that is considered to be life-supporting or life-sustaining and thus is essential to maintaining human life. Malfunctions of long-term implants are not routinely or "automatically" reportable unless the malfunction is likely to cause or contribute to a death or serious injury if it recurs.

The manufacturer takes or would be required to take an action under sections 518 or 519(f) of the Act as a result of the malfunction of the device or other similar devices. Conversely, malfunctions ARE NOT REPORTABLE if they are not likely to result in a death, serious injury, or another malfunction.

6.4.10 Government Contact Points

Where to obtain information:

- Consolidated Forms and Publications Distribution Center
- Beltsville Service Center
- 6351 Ammendale Road
- Beltsville, MD 20715

NOTE: Form FDA 3500A ONLY

- CDRH-Division of Industry and Consumer Education (DICE)
- Office of Communication and Education
- Center for Devices and Radiological Health
- Food and Drug Administration
- 10903 New Hampshire Avenue
- Silver Spring, MD 20993

1-800-638-2041

301-796-7100

DICE@fda.hhs.gov

- Food and Drug Administration
- MedWatch (HF-2)
- 5600 Fishers Lane, Room 17-65
- Rockville, MD 20857

1-800-FDA-1088 (Press "0" to speak with a staff member) or 301-827-7240

NOTE: FORM FDA 3500 ONLY Go to MedWatch and click on "How to Report".

- Reporting Systems Monitoring Branch
- Division of Surveillance Systems
- Office of Surveillance and Biometrics
- Center for Devices and Radiological Health
- 10903 New Hampshire Avenue, WO66
- Silver Spring, MD 20993

NOTE: FDA FORMS 3500A, 3417, and 3419 and instructions for each Web pages Medical Device Reporting (MDR) The instructions for the Mandatory MedWatch Form, 3500A.

WHERE TO SUBMIT ALL MANDATORY MDR REPORTS

- Food and Drug Administration
- Center for Devices and Radiological Health
- PO Box 3002
- Rockville, MD 20847-3002

NOTE: Envelopes must be specifically identified with the type of report enclosed, e.g., Manufacturer Report, User Facility Report, Baseline Report, Annual Report, Five-Day Report, Supplemental Report, etc.,

7 Access Controls

This document and supporting references establish controls for access to all Medical Data Networks LLC (MDN) software and any assemblies of software or configuration data supporting the T1Pal service (as a device).

This applies to all medical and research software devices designed and developed by MDN. Each of the key technologies for implementing electronic access controls is summarized below, consistent with VP and CEO requirements to implement comprehensive controls of access to all aspects of the T1Pal operations.

It is the policy that all employees shall be provided the most limited and/or most restrictive access to at most the most narrow set of T1Pal systems and data consistent with the tasks assigned to them based on their specific company role. In the event that such access requirements are temporary, access controls shall revert to their most restrictive state following temporary access use.

Specific access permissions are under control of the T1DPal CEO, plus one other VP officer. These individuals must specifically authorize all employee access settings. Access to all infrastructure controls requires a company-issued PC/Laptop desktop password and SSH access to all other T1Pal test and production components. Only the CEO may create and configure for use new cryptographic keys for use by others.

It is the duty of each and every associate of T1Pal to report any exceptions to this policy to the CEO, and to facilitate any corrective actions.

The following tools are currently in use by MDN so as to realize the above policy.

7.1 Subscriber Side Access Controls

1. **postgres Database**.

This database is referenced for comparison of encrypted subscriber-provided login and password credentials to credentials on record for such subscribers. The database is accessible only to certified other software components for query and retrieval of subscriber settings. Subscribers provide their credentials by submitting a form secured by SSL security encryption methods.

2. Semi-permanent network settings and network routing rules prevent subscribers from accessing data and/or configuration settings of any other subscriber.

3. Cryptographic keys generated by the T1Pal platform provide for subscriber-controlled sharing of read or read/write access to their own data.

4. Login actions taken by subscribers require authentication with Google's Single Sign On (oAuth) mechanism. T1Pal relies on that capability for all

user activity.

7.2 Network Operations-Side Software and Configuration Access Controls

GitHub. GitHub is the sole tool for storage of all permanent documents, drawings, agreements, and software.

All software for all components, regardless of their provenance or use, is stored and versioned in one or more named GitHub "repositories" owned by MDN and is under access control by its officers and designated employees.

This specifically includes all documents related to MDN's quality management system, and includes all open-sourced and/or commercial software components. GitHub also maintains a complete set of components applicable to the several different versions of software: Production, Staging, and Development.

Because GitHub is a distributed storage system, the computers used by all of the company officers, and software developers, and engineers is configured to store and access the complete set of GitHub repositories used by the firm. Access to these features are protected by two-factor authentication. Therefore, no specific backup resource or procedure –outside of GitHub– is needed or used when the number of copies of company Github repositories exceeds 2.

This approach enables each and every authorized employee and/or contractor to review the current, planned, and past revision history on all documents, software, and records.

7.3 Internal Network Operations Infrastructure Access Controls.

An exchange of cryptographic keys (SSH) is used to access all servers that are part of the T1Pal infrastructure. These cryptographic keys are themselves protected from access by strong userid/password credentials provided by the billing process of the infrastructure providers.

1. **Amazon Web Services (AWS)**. https://aws.amazon.com. AWS is one of the cloud providers used by MDN. Selected production MDN software is deployed to AWS. MDN also maintains non-production servers used to test software under development ("dev" environment) and ready to be deployed after acceptance testing ("staging environment).
2. **Digital Ocean**. Digital Ocean is one of the cloud providers used by MDN. Selected production MDN software is deployed to Digital Ocean. MDN also maintains non-production servers used to test software under development ("dev" environment) and ready to be deployed after acceptance testing and staging environment).

7.4 Corporate Administrative Access Controls

Google gmail addresses and credentials are used exclusively to provide intra-company communications.

All authoritative and/or permant documents are retained in the appropriate GitHub repository.

7.5 Backups and disaster recovery

1. All Google Drive contents are backed up by Google under contract with MDN.
   Two-factor authentication is required for all access.
2. All Google Drive and Google Vault backups are retained in perpetuity. Two-factor authentication is required for access.
3. All email is stored via Google Apps email and automatically backed up in Google Vault and is retained in perpetuity (including deleted emails). Two-factor authentication is required for access.
4. Redundant and duplicative distribution of GitHub-based repositories are retained on the primary computers of all officers, developers, and engineers. This eliminates the need for a separate and independent back up system. All of the software, documents, etc. are stored in the distributed GitHub-based system. Single-factor authentication is required for access. Two-factor authentication is required to read or publish changes in staging and production environments.

8 Gitflow Tailoring

This document describes the Gitflow procedure for developing and managing change to the existing software base of T1Pal.com The content and principles for this procedure is drawn from Vincent Driessen's Gitflow Workflow, herein incorporated as a reference for further detail.

This Gitflow Workflow defines a strict branching model designed around the project release. It relies on the functions and capabililities of "github."

All software and QMS documentation will be managed using this "Gitflow" procedures.

A diagram of the gitflow process is here: Git-branching-model.pdf

Key principles of this process are conventions described as follows:

1. The GITHUB repository for Medical Data Networks LLC (MDN) is distributed. As such, each and every developer and officer of the firm retains a complete copy of the MDN repositories on their laptop. This provides adequate storage, security, and reliability of all company records.
2. The firm's GITHUB server has as the "origin" a server assigned by github.com an application service provider.
3. The central repo holds two main branches with infinite lifetime:

- master
- dev

We consider origin/dev to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release. Some would call this the "integration branch". This is where any automatic nightly builds are built from.

When the source code in the develop branch reaches a stable point and is ready to be released, all of the changes should be merged back into master somehow and then tagged with a release number.

Next to the main branches master and develop, our development model uses a variety of supporting branches to aid parallel development between team members, ease tracking of features, prepare for production releases and to assist in quickly fixing live production problems. Unlike the main branches, these branches always have a limited life time, since they will be removed eventually.

The different types of branches we may use are:

- Feature branches
- Staging branches
- Hotfix branches

Each of these branches have a specific purpose and are bound to strict rules as to which branches may be their originating branch and which branches must be their merge targets.

Feature branches (or sometimes called topic branches) are used to develop new features for the upcoming or a distant future release. When starting development of a feature, the target release in which this feature will be incorporated may well be unknown at that point. The essence of a feature branch is that it exists as long as the feature is in development, but will eventually be merged back into develop (to definitely add the new feature to the upcoming release) or discarded (in case of a disappointing experiment).

Feature branches typically exist in developer repos only, not in origin.

8.1 Creating a feature branch

When starting work on a new feature, branch off from the develop branch.

$ git checkout -b myfeature develop

Switched to a new branch “myfeature”

##Incorporating a finished feature on develop

Finished features may be merged into the develop branch to definitely add them to the upcoming release:

$ git checkout develop

Switched to branch ‘develop’

$ git merge –no-ff myfeature

Updating ea1b82a..05e9557

(Summary of changes)

$ git branch -d myfeature

Deleted branch myfeature (was 05e9557).

$ git push origin develop

The –no-ff flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature.

8.2 Staging branches

- May branch off from:
- dev

- Must merge back into:
- dev and master
- Branch naming convention:
- release-*

Release branches support preparation of a new production release. They allow for last-minute dotting of i's and crossing t's. Furthermore, they allow for minor bug fixes and preparing meta-data for a release (version number, build dates, etc.). By doing all of this work on a release branch, the develop branch is cleared to receive features for the next big release.

The key moment to branch off a new release branch from develop is when develop (almost) reflects the desired state of the new release. At least all features that are targeted for the release-to-be-built must be merged in to develop at this point in time. All features targeted at future releases may not—they must wait until after the release branch is branched off.

It is exactly at the start of a release branch that the upcoming release gets assigned a version number—not any earlier. Up until that moment, the develop branch reflected changes for the "next release", but it is unclear whether that "next release" will eventually become 0.3 or 1.0, until the release branch is started. That decision is made on the start of the release branch and is carried out by the project's rules on version number bumping.

##Creating a release branch

Release branches are created from the develop branch. For example, say version 1.1.5 is the current production release and we have a big release coming up. The state of develop is ready for the "next release" and we have decided that this will become version 1.2 (rather than 1.1.6 or 2.0). So we branch off and give the release branch a name reflecting the new version number:

$ git checkout -b release-1.2 develop

Switched to a new branch "release-1.2"

$ ./bump-version.sh 1.2

Files modified successfully, version bumped to 1.2.

$ git commit -a -m "Bumped version number to 1.2"

[release-1.2 74d9424] Bumped version number to 1.2

1 files changed, 1 insertions(+), 1 deletions(-)

After creating a new branch and switching to it, we bump the version number. Here, bump-version.sh is a fictional shell script that changes some files in the working copy to reflect the new version. (This can of course be a manual change—the point being that some files change.) Then, the bumped version number is committed.

This new branch may exist there for a while, until the release may be rolled out definitely. During that time, bug fixes may be applied in this branch (rather than on the develop branch). Adding large new features here is strictly prohibited. They must be merged into develop, and therefore, wait for the next big release.

8.3 Finishing a release branch

When the state of the release branch is ready to become a real release, some actions need to be carried out. First, the release branch is merged into master (since every commit on master is a new release by definition, remember). Next, that commit on master must be tagged for easy future reference to this historical version. Finally, the changes made on the release branch need to be merged back into develop, so that future releases also contain these bug fixes.

The first two steps in Git:

$ git checkout master

Switched to branch 'master'

$ git merge –no-ff release-1.2

Merge made by recursive.

(Summary of changes)

$ git tag -a 1.2

The release is now done, and tagged for future reference.

To keep the changes made in the release branch, we need to merge those back into develop, though. In Git:

$ git checkout develop

Switched to branch 'develop'

$ git merge –no-ff release-1.2

Merge made by recursive.

(Summary of changes)

This step may well lead to a merge conflict (probably even, since we have changed the version number). If so, fix it and commit.

Now we are really done and the release branch may be removed, since we don't need it anymore:

$ git branch -d release-1.2

Deleted branch release-1.2 (was ff452fe).

8.4 Hotfix Branches

May branch off from:

master

Must merge back into:

develop and master

Branch naming convention:

hotfix-*

Hotfix branches are very much like release branches in that they are also meant to prepare for a new production release, albeit unplanned. They arise from the necessity to act immediately upon an undesired state of a live production version. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.

The essence is that work of team members (on the develop branch) can continue, while another person is preparing a quick production fix.

8.5 Creating the hotfix branch

Hotfix branches are created from the master branch. For example, say version 1.2 is the current production release running live and causing troubles due to a severe bug. But changes on develop are yet unstable. We may then branch off a hotfix branch and start fixing the problem:

$ git checkout -b hotfix-1.2.1 master

Switched to a new branch “hotfix-1.2.1”

$ ./bump-version.sh 1.2.1

Files modified successfully, version bumped to 1.2.1.

$ git commit -a -m “Bumped version number to 1.2.1”

[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1

1 files changed, 1 insertions(+), 1 deletions(-)

Don’t forget to bump the version number after branching off!

Then, fix the bug and commit the fix in one or more separate commits.

$ git commit -m “Fixed severe production problem”

[hotfix-1.2.1 abbe5d6] Fixed severe production problem

5 files changed, 32 insertions(+), 17 deletions(-)

8.6 Finishing a hotfix branch

When finished, the bugfix needs to be merged back into master, but also needs to be merged back into develop, in order to safeguard that the bugfix is included in the next release as well. This is completely similar to how release branches are finished.

First, update master and tag the release.

$ git checkout master

Switched to branch ‘master’

$ git merge –no-ff hotfix-1.2.1

Merge made by recursive.

(Summary of changes)

$ git tag -a 1.2.1

Edit: You might as well want to use the -s or -u flags to sign your tag cryptographically.

Next, include the bugfix in develop, too:

$ git checkout develop

$ git merge –no-ff hotfix-1.2.1

Merge made by recursive.

8.7 Summary of changes

The one exception to the rule here is that, when a release branch currently exists, the hotfix changes need to be merged into that release branch, instead of develop. Back-merging the bugfix into the release branch will eventually result in the bugfix being merged into develop too, when the release branch is finished. (If work in develop immediately requires this bugfix and cannot wait for the release branch to be finished, you may safely merge the bugfix into develop now already as well.)

Finally, remove the temporary branch:

$ git branch -d hotfix-1.2.1

Deleted branch hotfix-1.2.1 (was abbe5d6).

9 Semantic Versioning

Semantic Versioning is the well-defined process of assigning version numbers to software products that expose Application Programming Interfaces (APIs). It

is a set of rules and requirements that dictate how version numbers are assigned and incremented.

It is a consistent approach that saves time and cost otherwise needed to support release and upgrade packages without having to roll new versions of dependent packages.

Prefixing a semantic version with a "v" is a common way (in English) to indicate it is a version number. Abbreviating "version" as "v" is often seen with version control. Example: git tag v1.2.3 -m "Release version 1.2.3", in which case "v1.2.3" is a tag name and the semantic version is "1.2.3".

Semantic Versioning is also dependent on accurate and complete API definitions.

It is hereby adopted as a standard means of conveying robust dependency information for all T1Pal services. For reference purposes the github repository defining Semantic Versioning is available.

Semantic Versioning is used for all software artifacts produced or used by Medical Data Networks LLC.
This document provides the guidance for this usage, much of which is copied within this QMS from the link above.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Software using Semantic Versioning MUST declare a public API. This API could be declared in the code itself or exist strictly in documentation. However it is done, it SHOULD be precise and comprehensive.

A normal version number MUST take the form X.Y.Z where X, Y, and Z are non-negative integers, and MUST NOT contain leading zeroes. X is the major version, Y is the minor version, and Z is the patch version. Each element MUST increase numerically. For instance: 1.9.0 -> 1.10.0 -> 1.11.0.

Once a versioned package has been released, the contents of that version MUST NOT be modified. Any modifications MUST be released as a new version.

Major version zero (0.y.z) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable.

Version 1.0.0 defines the public API. The way in which the version number is incremented after this release is dependent on this public API and how it changes.

Patch version Z (x.y.Z | x > 0) MUST be incremented if only backwards compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.

Minor version Y (x.Y.z | x > 0) MUST be incremented if new, backwards compatible functionality is introduced to the public API. It MUST be incremented if any public API functionality is marked as deprecated. It MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes. Patch version MUST be reset to 0 when minor version is incremented.

Major version X (X.y.z | X > 0) MUST be incremented if any backwards incompatible changes are introduced to the public API. It MAY also include minor and patch level changes. Patch and minor version MUST be reset to 0 when major version is incremented.

A pre-release version MAY be denoted by appending a hyphen and a series of dot separated identifiers immediately following the patch version. Identifiers MUST comprise only ASCII alphanumerics and hyphens [0-9A-Za-z-]. Identifiers MUST NOT be empty. Numeric identifiers MUST NOT include leading zeroes. Pre-release versions have a lower precedence than the associated normal version. A pre-release version indicates that the version is unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version. Examples: 1.0.0-alpha, 1.0.0-alpha.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92, 1.0.0-x-y-z.–.

Build metadata MAY be denoted by appending a plus sign and a series of dot separated identifiers immediately following the patch or pre-release version. Identifiers MUST comprise only ASCII alphanumerics and hyphens [0-9A-Za-z-]. Identifiers MUST NOT be empty. Build metadata MUST be ignored when determining version precedence. Thus two versions that differ only in the build metadata, have the same precedence. Examples: 1.0.0-alpha+001, 1.0.0+20130313144700, 1.0.0-beta+exp.sha.5114f85, 1.0.0+21AF26D3—-117B344092BD.

Precedence refers to how versions are compared to each other when ordered.

Precedence MUST be calculated by separating the version into major, minor, patch and pre-release identifiers in that order (Build metadata does not figure into precedence).

Precedence is determined by the first difference when comparing each of these identifiers from left to right as follows: Major, minor, and patch versions are always compared numerically.

Example: 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1.

When major, minor, and patch are equal, a pre-release version has lower precedence than a normal version:

Example: 1.0.0-alpha < 1.0.0.

Precedence for two pre-release versions with the same major, minor, and patch version MUST be determined by comparing each dot separated identifier from left to right until a difference is found as follows:

Identifiers consisting of only digits are compared numerically.

Identifiers with letters or hyphens are compared lexically in ASCII sort order.

Numeric identifiers always have lower precedence than non-numeric identifiers.

A larger set of pre-release fields has a higher precedence than a smaller set, if all of the preceding identifiers are equal.

Example: 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha.beta < 1.0.0-beta < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 < 1.0.0.

10 Configuration Management

10.1 Preferred Use over Command Lines

The use of configuration files is intended to eliminate and/or reduce the use of command line functions to control and configure the components of T1Pal operations. The reduction and/or elimination of command-line functions is a key to automating complex tasks that require the attention of highly skilled associates at inconvenient hours.

10.2 Scope of Use

Configuration files shall be used in every software design used and developed by Medical Data Networks LLC (MDN). In addition, all such configuration files shall be managed as code within the github software repository.

Configuration files will be read upon the deployment and/or restart of new software components.
Configuration files shall themselves be versioned (refer to Semantic Versioning guidelines).

11 Performance Management

For all software services provided via API, "health check" probes shall be provided to measure the response time of the API under load. The response time should include measures of internal "real-world" function of the API, preferably using test credentials that fully test database connectivity, database functions, and health checks of subtending API services.

Time series logs of health check calls shall be logged for the most recent 30 days. Tools to use the logs to display charts and graphs shall be provided with the deployment of new API versions.

11.1 References

1. 21 CFR 820
2. FDA
3. Quality System Regulation
4. ISO 13485:2016 Clause 4.2.5

11.2 Requirements

MDN shall provide a dashboard showing the current performance status of the T1Pal operation. The intent is to minimize wasted time spent debugging faults that impact the user experience.

12 Billing Management

MDN shall conduct its product selection process, invoicing, and other billing operations so as to maintain privacy requirements of both HIPAA and PCI (payment card industry) demands. The following sections identify how this is accomplished.

12.1 Credit Card Service Operations

"Stripe.com" and "Servicebot.io" are providers of billing services to T1Pal operations. These providers are separate and independent companies that provide all of the credit card and related payment services to T1Pal software operations. All communications between T1Pal, and subscribers, to/from these providers is encrypted using standard SSL methods.

There are functions of stripe.com that fail to completely meet PCI demands for credit card handling. This issue is well known and was created when stripe.com introduced advanced analysis of stripe.com customer behaviors in an "SQL" engine. The remedy for these has been accepted and planned by stripe.com.

Generally, processing payments through a credit card processor creates personally identifiable information (PII) that must be secured both in transmission and storage. However, US Health and Human Services (HHS) have stated that collecting payments is excluded explicitly from HIPAA mandates.

For T1Pal to continue to utilize Stripe for invoicing, and financial analysis, T1Pal will need to get a Business Associate Agreement (BAA) from Stripe, and configure use of Stripe to comply with HIPAA requirements. A BAA template is expected in follow-on revisions to this QMS.

In addition, Payment Card Industry Data Security Standards (PCI DSS). sets the minimum standard for credit card data security, and requires a periodic audit of security operations. T1Pal expects to achieve PCI Level 4 compliance, which includes use of a checklist and an annual audit. Both the checklist and actual audit results will be managed in revisions to this QMS.

13 Design, Release and Operations Management

T1Pal Code Repository and the Gitflow Process

It is the standard procedure to use github.com as a service provider for software. In addition, it is the standard procedure to follow the "Gitflow" method for managing improvements to software targeted for the T1Pal platform. Gitflow as tailored to meet T1Pal requirements is further described in the QMS SOP_0008_Gitflow_Tailoring.md section of this QMS system.

Any and all software released as part of T1Pal shall first be introduced to (i.e. created on) the T1Pal platform as a branch of the T1Pal github code repository. All software is created first on either of the following two branches:

1. "**feature**" branch – for software not yet associated with a relase.
2. "**development**" branch – for software definitely targeted to a particular relase.

Following the "**gitflow**" process, all T1Pal "code" goes into this github code repository.
By "code" we include the following artifacts:

- software of any particular language (javascript, Ruby, PHP, BASH)
- configuration files (e.g. nginx, apache web server configuration files)
- service definition files (e.g. docker files, .yaml files for Kubernetes, HOLM charts)
- Instructions for manually configuring services from a particular platform service provider (e.g. Digital Ocean servers, AWS, MongoDB, Redis, postgresdb)

Component services built on Docker containers shall be "built" using Docker files controled by 1) a docker command-line or 2) a script.

In either case, the command-line and/or the script shall itself be included in the development github repository. Configuration parameters for all functions of a docker container shall be provided within the Docker file, and stored in the github repository.

A specific BASH script shall be used to trigger the transfer and activation of new product instances.

T1Pal uses a number of external services that are configured manually and then operationally accessed using a well-defined API. These including stripe.com, servicebot.io, MongoDB, postgresdb, DNS, Kubernetes clusters, and Digital Ocean hosting services. It is the goal of T1Pal to leverage all such public services to the maximum extent possible, subject to operational cost, reliability, security, and performance requirements.

Step-by-step procedures to manually create and utilize any external service shall be documented in the T1Pal development github repository in .md format.

Release Process A "release" of new or changed software to production amounts to promoting the "staging" branch of the github repository to the "master" branch of the github repository, where it can be tested prior to final release.

The production branch of the repository "should" exactly match the deployed production software.

The release process generally involves logging into a production server so as to remotely "check out" the correct development branch from the github repository. Each component of the T1Pal system has it's own "repo" that has a development, staging, feature, and release branches.

The "staging" branch of the github repository contains the tested final version to be promoted to production. The "staging" branch is subject to extensive testing and is only available to operations staff.

A "run-book" is a check-list style document (stored in the github repository) that provides all of the methods and procedures needed to promote a "staging" branch to the "production" branch and test for correct operation. Testing of the run-book is also done in the staging environment.

Prior to checking out the production software on a T1Pal component (from the master repo), a data transition script may be required to prepare the production environment with the right data, in the right schema. A backup of the production data must be done for all software changes impacting data.

13.1 General Design and Development Planning Processes

It is the policy of T1Pal to align with "FDA 21 CFR 820.30 (b)" with respect to design and development planning:

"Each manufacturer shall establish and maintain plans that describe or reference the design and development activities and define responsibility for implementation.

The plans shall identify and describe the interfaces with different groups or activities that provide, or result in, input to the design and development process.

The plans shall be reviewed, updated, and approved as design and development evolves."

MDN uses an iterative, agile development process. Design planning and development is an ongoing activity; all related activity is captured and documented on the corporate github repository. Planning, design input, design output, review and testing activities may all occur at any time. These activities are described below in Section 7, Design and Development Activities.

1. Design and Development **Planning Sessions**. Design and development planning happens during planning meetings, which may occur at any time. All decisions about what work is to be done in the sprint are captured in GitHub a issues folder. The results of the planning sessions are documented on GitHub.
2. **Requirements**. Requirements for a given piece of functionality are documented in a GitHub issue. Each issue has a unique URL. The requirement is typically summarized in the title of the issue or within the issue's description.
3. **User interface design**. User Interface designs are documented and stored in Google Drive. All designs are available for public inspection. Only MDN employees can make changes to User Interface design documents. All revisions to design documents are maintained indefinitely by Google Drive and available at all times.
4. **Software design**. Where appropriate, software designs, including revisions to the designs, are documented in GitHub in the relevant repository. Other architecture documents may be stored in GitHub or Google Drive and will be linked to Jira issues. Software design may also be documented in the source code itself. All software design documentation including all revisions is maintained indefinitely in GitHub and Google Drive.

Most GitHub repositories represent an area of software functionality that is exposed to end users. Some repositories provide common functionality or test frameworks that may not be exposed to end users.

The README file in each repository provides an overview of that repositories' purpose or functionality.

The Master branch for each repository represents the latest software source code updates to that functionality. GitHub Tags are used to uniquely identify specific checkpoints of functionality; not all tags are deployed to production. For most products, tags are given semantic version numbers following this pattern:

MAJOR.MINOR.PATCH, where

- MAJOR version indicates changes for major new functionality or incompatible API changes,
- MINOR version indicates when new functionality is added in a backwards-compatible manner, and
- PATCH version indicates when backwards-compatible bug fixes are made.

For some products (such as the MDN Uploader), tags may only use MAJOR.MINOR numbering. In these cases, patch releases will cause the MINOR number to increment.

Software is typically only deployed from a tag created on the Master branch of a repository. Only in the rare case of an urgent/important "hotfix" will be a

deployment be made from a tag created on a branch other than Master. That fix will later be incorporated into Master.

Software Development

Software developers develop, modify and test software on their local development workstation.

Software changes can only be committed to the GitHub Master branch by a MDN employee or contractor (not by an open source developer).

For all non-trivial changes, the software developer must seek review from another developer (known as reviewing a "Pull Request") prior to committing to Master.

1. **Design History File (DHF)** The combination of Jira,, Google Docs, Zendesk, and GitHub together comprises the Design History File. See section 14 for details of the DHF.
2. **Development Responsibility** The CEO is responsible for the appropriateness, quality, processes and implementation of all development activities.
3. **Interfaces with Design Input.** All employees, contractors and consultants at MDN may interact with Design Input sources. Those sources are detailed in Section 8, Design Input.

Updates.

1. **Regular updates**. MDN software may be deployed at any time. Verification activities are conducted for each Jira issue (see sections 11 and 12 below). Validation activities are conducted for each new product released.
2. **Approval and Deployment (Design Transfer to Production)**. Deployment to production of customer-facing functionality must be approved by a VP-level employee or the CEO.

13.2 Design Input

Consistent with 21 CFR 820.30 (c), each manufacturer shall establish and maintain procedures to ensure that the design requirements relating to a device are appropriate and address the intended use of the device, including the needs of the user and patient. The procedures shall include a mechanism for addressing incomplete, ambiguous, or conflicting requirements. The design input requirements shall be documented and shall be reviewed and approved by a designated individual(s). The approval, including the date and signature of the individual(s) approving the requirements, shall be documented.

It is MDN's policy and practice to capture designs in the form of two key artifacts: * Sequence Diagrams

MDN has in its repository a collection of UML-style Sequence Diagrams showing the principal flows among all of the relevant components for each subflow. These

diagrams, when annoted with API and other reference information establishes both the current and planned design of the entire T1Pal system.

- Swagger API Definitions for RESTful interfaces.

Standard RESTful APIs are the preferred means of inter-system communications. These are best documented in the form of "Swagger" files that completely define the API.

13.2.1 Sources of Design Inputs.

Input into the MDN design and development process, including requirements, may come from these sources, but is not limited to these sources:

- Interviews with actual and potential users MDN conducts in person, phone, video conference and email interviews with actual and potential users. The notes from these meetings are stored in Google Drive as Google Documents. Users include but are not limited to: people with diabetes, their care team, parents, doctors, diabetes educators, device makers and diabetes researchers.
- MDN Employees and Contractors**: MDN's employees, contractors and consultants may also be the source of design input.
- Observed and measured use of MDN software: When running, MDN collects metrics and usage statistics. These data may also be sources of design input.
- Feedback from Alpha and Beta Users: MDN collects feedback from Beta users as part of Software Validation activities (see section 11). Results from these activities may also be used as design input.
- Feedback received via social media, such as Twitter, Facebook, Instagram and comments on blog posts.
- Feedback from users via customer support, including emails to support@tidepool.org and via our support ticketing system.
- Appropriateness of requirements to meet intended use: It is the responsibility of the VP of Product or CEO to approve the appropriateness of product functionality that is made available via production servers. MDN developers may deploy code to non-production servers (such as "staging", "development" or "integration" servers) without VP or CEO approval.
- Mechanism for addressing incomplete, ambiguous, or conflicting requirements: Incomplete, ambiguous or conflicting requirements will be discussed during planning meetings and via ongoing conversation. It is the responsibility of the VP of Product or CEO to resolve incomplete, ambiguous or conflicting requirements for product functionality that is made available via production servers. The outcome of resolving these requirements issues is documented in Jira issues.

- Review and approval: Documentation for review and approval of all functionality is captured on Jira issues.

13.3 Design Output

- Reference 21 CFR 820.30 (d): Each manufacturer shall establish and maintain procedures for defining and documenting design output in terms that allow an adequate evaluation of conformance to design input requirements. Design output procedures shall contain or make reference to acceptance criteria and shall ensure that those design outputs that are essential for the proper functioning of the device are identified. Design output shall be documented, reviewed, and approved before release. The approval, including the date and signature of the individual(s) approving the output, shall be documented.

UML-styled Sequence Diagrams described above are used to capture design Outputs unambiguously and subject ot change control documentation.

- Evaluation of conformance to requirements.
- Verification and deployment of software: Prior to deployment to production servers, software is deployed to a staging server environment where it is tested. The QA department is responsible for developing and executing verification tests (see section 10) although others including volunteer testers, other employees, and the CEO may also execute tests.
- Documentation:
- All verification tests, both templates and executed tests, are documented in Google Docs and stored indefinitely in Google Drive, including a record of all changes to each file.
- Deployment of software to development, staging and production server environments is documented as a Jira list for each deployment.
- Review and approval**. The QA department is responsible for review and approval of all verification tests. VP-level or CEO approval is required for deployment to production servers.

13.4 Design Review

- User interface reviews: User interface designs are reviewed via Google Drive. Comments on designs in Google Drive are maintained in perpetuity.
- Code reviews: All substantial changes to software source code intended to be deployed to production must be reviewed by a MDN employee other than the author of the source code. Review of design results.
- Review by individuals without direct responsibility:

- UI Design Review. All UI designs are reviewed by someone other than the UI designer.
- Code Review. An MDNs employee or contractor is responsible for reviewing all substantial code changes that will result in customer-facing functionality. The employee who proposed the changes (via GitHub pull request) may not review the proposed changes.

13.5 Documentation

All relevant reviews are documented in Jira or Google Docs and stored in perpetuity in Google Drive. All changes and records, including the date and individuals performing the review, are maintained in perpetuity.

13.6 Design Verification

- Black Box Test Templates: Black Box" testing refers to test verification activities that are performed manually, not by automated code.
- Location, name and version: Test verification templates are stored in Google Drive in folders with a unique name indicating the area being tested and the version number of the test.
- Responsibility for tests: Any MDN employee, contractor or consultant can create new "Black Box" tests.
- Changes to tests: A record of changes to tests are maintained in perpetuity in the document version history for each test. Obsolete tests are prominently marked [OBSOLETE] and are moved to an archive directory.
- Executed tests: For executed tests, the test template is copied to the "Executed Verification Scripts" folder where it is edited as the test is executed. Elements of the test are marked "PASSED" or "FAILED." For elements of the test that have failed, a Jira issue is generated to address the issue.
- Automated "White Box" tests: At the discretion of the software developer, software functionality may be tested using automated tests.
- Software source code for the automated test suite will be stored in the same GitHub repository as the software functionality itself.
- If functionality includes an automated test or suite of tests, all tests must pass for the software source code can be merged into the "master" branch.

13.7 Documentation

- Methods: The method of verification is documented in the test verification template per Section 10.1. Links to each documented test are included in the DHF as described in Section 14.

- Date and Responsibility: The date of running the verification test as well as the individuals performing the test are detailed in the executed test document per Section 10.2.
- Design Validation
- Defined operating conditions: MDN software is intended to be used on computers and mobile devices with an Internet connection.
- Defined user needs and intended use: User needs are documented in requirements as per Section . The intended uses of MDN software are defined.
- Software validation: The use of MDN software is validated by volunteer
- Beta users. Beta users are selected to based on expressed interest in testing MDN software, configurations they have available (e.g. Mac or Windows, which diabetes devices they use), and demonstrated ability to complete homework assignments. Beta users are given homework assignments with instructions on how to test software functionality. Homework assignment templates are stored in the "Validation Templates" folder. Beta users document both quantitative and qualitative results of their use of the software in Google Docs. Those Google Docs are stored in Google Drive in the "Validation Documentation" folder. MDN software is tested by Beta users with their own computers and mobile devices in their own actual use conditions.

13.8 Risk analysis

A Risk Analysis review, consistent with chapter 4 "Risk and Hazard Management" is required before commencing Design Transfer.

13.9 Design Transfer

- Consistent with 21 CFR 820.30 (h) each manufacturer shall establish and maintain procedures to ensure that the device design is correctly translated into production specifications.
- Transfer of development code to production. A software developer is assigned during the Sprint Planning meeting and identified on the github issue for implementation of the task. It is the responsibility of the software developer to implement the software per requirements detailed on the Jira issue or via UI design in Google Drive. If there are ambiguous requirements or designs, the ambiguity may be resolved through conversation. For substantial ambiguity, resolution will be documented in the Jira card. For minor changes or ambiguity, the resolution may be documented in the software code itself.

- Responsibility and Approval**. It is the responsibility of the VP of Product and/or CEO to assure that all requirements have been met, that designs have been correctly translated into Production, and that all appropriate verification and validation activities have been performed. VP of Product or CEO approval is required for deployment to Production of all substantial, customer-facing functionality.

Documentation. All development transfer activities are documented in githugithubb. For deployments to Production servers, a new Jira list will be created that contains the name, date and version number of software being deployed.

13.10 Design Changes

- Reference 21 CFR 820.30 (i) Design changes: Each manufacturer shall establish and maintain procedures for the identification, documentation, validation or where appropriate verification, review, and approval of design changes before their implementation.
- Iteration of Design Changes: MDN uses an agile development model. Design changes may occur in any sprint. In addition to documenting design as described above in Section 7, Design and Development Activities, ongoing changes to design will be documented as described below.
- Identification: All documents in GitHub are uniquely identified by a unique Uniform Resource Identifier (URI) that is assigned by the respective service. Documents may additionally have a human-readable document title and document ID. For example, this document is:

13.11 Documentation of design changes.

- All changes to requirements, design, software source code, functionality, verification and validation are documented and tracked in Jira, Pixelapse, GitHub and Google Docs. Revision history is maintained in perpetuity.
- Changes to requirements are identified and documented in github issues. A history of changes to github issues is maintained in perpetuity.
- Changes to software source code are maintained in GitHub. Revision history is maintained in perpetuity.
- Review and Approval: Design changes are reviewed and approved before deployment to production. All review and approval will be documented in Jira, Pixelapse, GitHub or Google Docs.

13.12 Design History File (DHF)

- Reference 21 CFR 820.30 (j) Design history file: Each manufacturer shall establish and maintain a DHF for each type of device. The DHF shall

contain or reference the records necessary to demonstrate that the design was developed in accordance with the approved design plan and the requirements of this part.
- Together, the documentation maintained in GitHub comprises each MDN product's Design History File (DHF).

13.13 CFR Reference: 21 CFR 820.30 Design Controls

- Design and development planning: Each manufacturer shall establish and maintain plans that describe or reference the design and development activities and define responsibility for implementation. The plans shall identify and describe the interfaces with different groups or activities that provide, or result in, input to the design and development process. The plans shall be reviewed, updated, and approved as design and development evolves.

- Design input: Each manufacturer shall establish and maintain procedures to ensure that the design requirements relating to a device are appropriate and address the intended use of the device, including the needs of the user and patient. The procedures shall include a mechanism for addressing incomplete, ambiguous, or conflicting requirements. The design input requirements shall be documented and shall be reviewed and approved by a designated individual(s). The approval, including the date and signature of the individual(s) approving the requirements, shall be documented.

- Design output: Each manufacturer shall establish and maintain procedures for defining and documenting design output in terms that allow an adequate evaluation of conformance to design input requirements. Design output procedures shall contain or make reference to acceptance criteria and shall ensure that those design outputs that are essential for the proper functioning of the device are identified. Design output shall be documented, reviewed, and approved before release. The approval, including the date and signature of the individual(s) approving the output, shall be documented.

- Design review: Each manufacturer shall establish and maintain procedures to ensure that formal documented reviews of the design results are planned and conducted at appropriate stages of the device's design development. The procedures shall ensure that participants at each design review include representatives of all functions concerned with the design stage being reviewed and an individual(s) who does not have direct responsibility for the design stage being reviewed, as well as any specialists needed. The results of a design review, including identification of the design, the date, and the individual(s) performing the review, shall be documented in the design history file (the DHF).

- Design verification: Each manufacturer shall establish and maintain procedures for verifying the device design. Design verification shall confirm

that the design output meets the design input requirements. The results of the design verification, including identification of the design, method(s), the date, and the individual(s) performing the verification, shall be documented in the DHF.

- Design validation: Each manufacturer shall establish and maintain procedures for validating the device design. Design validation shall be performed under defined operating conditions on initial production units, lots, or batches, or their equivalents. Design validation shall ensure that devices conform to defined user needs and intended uses and shall include testing of production units under actual or simulated use conditions. Design validation shall include software validation and risk analysis, where appropriate. The results of the design validation, including identification of the design, method(s), the date, and the individual(s) performing the validation, shall be documented in the DHF.
- Design transfer: Each manufacturer shall establish and maintain procedures to ensure that the device design is correctly translated into production specifications.
- Design changes: Each manufacturer shall establish and maintain procedures for the identification, documentation, validation or where appropriate verification, review, and approval of design changes before their implementation.
- Design history file: Each manufacturer shall establish and maintain a DHF for each type of device. The DHF shall contain or reference the records necessary to demonstrate that the design was developed in accordance with the approved design plan and the requirements of this part.

13.14 3. References

1. U.S. Food and Drug Administration, 21 CFR Part 820: Quality System Regulation.
2. General Principles of Software Validation; Final Guidance for Industry and FDA Staff
3. AAMI TIR45:2012 Technical Information Report Guidance on the use of AGILE practices in the development of medical device softwareOverview

Operations Management

The T1Pal solution is in fact an interworking set of services that are each somewhat independent of each other. The run-book described above is used to direct operations staff to reports, displays of data flows, lists of resources consumed, and response times for specified components.

The operations staff must monitor all displays and reports continuously 24 x 7 x 365 and at the same time monitor trouble ticket complaints.

In the event of an outage of any kind, the outage must be documented on freshstatus.com web site so as to alert customers subscribing to that service. https://t1pal-995.freshstatus.io/ Any and all observed impairments to any T1Pal components should be entered into a trouble ticket. The trouble ticket system is configured to receive emails directed to support@t1pal.com The trouble ticketing system is at t1pal.helpy.io

13.15 Purpose

This file contains an overview of the Medical Data Networks LLC Product Development Process. It overlaps somewhat with the SOP_0008_Gitflow_Tailoring.md under the SOP document folder.

The Product Development Process is by definition the process by which changes to software and services that are used in the T1Pal application and its supporting infrastructure.

The complete Product Developmetn Process is composed of 5 phases. + Phase 1: Research and select specific changes to Standard Operating Procedures to capture in a modified system design. The deliverable for this phase is the requirements for the change to be made. + Phase 2: Develop Software changes by modifying, deleteing, and/or adding code. There are 6 steps in this phase: + Cut a new feature branch for the change and do development on that feature branch. + When ready, the developer puts the code on a shared developer branch (DEV) for team review and automated testing. + Automated testing happens, triggered by placement of developed code onto the DEV branch. + Technical reviewers sign-off on changes based on test findings. + Changed software is pushed to master branch (per gitflow process) + Cut a new software relase to production using semantic versioning.

- Phase 3: Maintenance Operations
  - Review and adjust Fault Management solution (health checks)
  - perform backups of any impacted dataset.
  - perform system health checks
  - validate events detected
  - fix or roll back software
- Phase 4: Release changes to production
  - follow the release process to deploy code.
  - if needed, update the SOP
  - if needed, update the support knowledge base.
- Phase 5: Post Market Surveillance
  - address all of the outstanding bugs reported and assess relative value
  - address “pain-points” of current and prospective users.
  - address the features needed to enter new, adjacent markets
  - address features in the value chain (above and below) to address cost reductions and performance improvements.

General This is our repository at MDN for general developer/technical docu-

mentation that doesn't belong in a specific app/library/service repo. Table of contents Here in the docs repository: • Code Peer Review Checklist • Development Process • GitHub Processes • External Code Dependency Considerations • External Service API Dependency Considerations

Tidepool Code Peer Review Checklists

In order to consistently ensure high quality code, a peer review of all code changes is required. It is recommended that the developer actually perform a code review of their own code before submitting it for peer review. This will hopefully reduce the amount of time and effort spent by the reviewer by catching common issues beforehand. This document is intended as guidance for engineers. It is not a Standard Operating Procedure (SOP). Common Checklist 1. Does the code complete the feature requirements? Does it match the Jira issue? 2. Is the feature branch up-to-date with the develop branch? 3. Does the continuous integration build (e.g. Travis, CircleCI) pass? Do any of the non-required continuous integration steps fail? If so, why? Can these be fixed? If not, why not? 4. If any external dependencies were added: i. Review the results of the MDN External Dependency Considerations for each new dependency and sub-dependency. ii. Do all of the new dependencies and sub-dependencies use an acceptable license? iii. Are the new dependencies and sub-dependencies properly versioned and/or vendored to allow for future repeatable builds? 5. If any external dependencies were updated: i. Was each update explicit? ii. Were any updates accidental or unintentional? 6. Does all new or updated code have a minimum level of test coverage? 7. Are critical edge cases properly tested? 8. Is overall test coverage at least the same or improved? 9. Visually review all new and updated code, with particular focus on the non-test code. i. Does it generally follow good industry practices, such as: a. Separation of concerns b. SOLID c. Single responsibility principle - a class should have only a single responsibility (i.e. only changes to one part of the software's specification should be able to affect the specification of the class) d. Open/closed principle - software entities should be open for extension, but closed for modification e. Liskov substitution principle - objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program f. Interface segregation principle - many client-specific interfaces are better than one general-purpose interface g. Dependency inversion principle - one should depend upon abstractions, not concretions h. DRY - Don't Repeat Yourself i. YAGNI - You Ain't Gonna Need It (did you gold plate it?) j. Principle of least privilege - minimum needed to get the job done ii. Does it conform to the latest best-practices for that repository, language, and/or framework (e.g. format, lint, import)? a. Naming conventions (CamelCase, UPPERCASE, etc.)? b. Spaces vs. tabs and indentation, as appropriate for the repository. iii. Is it reasonably self-documenting? a. Variable names should be descriptive. Use of single-letter variable names are typically only valid for loop index variable names. iv. For complex algorithms, is the algorithm documented through more than just the code (i.e. comments)? v. Are all returned errors and exceptions checked? vi. Are all returned errors and exceptions properly handled? Depending upon the

situation this could mean propagating the error to the caller and/or reporting the error to the log. a. Are all resources properly disposed even when an error occurs (e.g. files actually closed if there is a failure while reading, temporary files deleted). vii. Do all switch statements handle the default case properly? viii. Do all for loops correctly manage memory and other resources? a. Resources opened within a loop usually need to be confirmed closed in the loop. b. Memory allocations within a loop may not be garbage collected until well after the loop completes. c. Consider unrolling or breaking up problematic loops. ix. If any code was copy-pasted: a. Why? Could the code have been reused instead with a little effort? b. If not, were any subtle bugs introduced with the copy-paste? x. Is it more complicated than it needs to be? Can it be made simpler? Easier to read or understand? xi. Does it use framework features wherever possible? No sense in reimplementing standard library features. xii. Are there extra features that might get used someday? If so, consider removing the extra, unused code. xiii. Are there any functions or methods that are particularly long (e.g. more than 30 lines or so)? Should they be broken down into separate, more focused functions that are easier to understand and test? xiv. Is there any newly comment-out code? Any code not being used should be removed. Under certain uncommon circumstances it is okay to leave commented-out code, but there should be detailed documentation before the block that describes why it is commented out and yet still remains in the code. xv. Are there any special values or hard-coded values? If so, they should probably be constants or enums, especially if shared across modules/interfaces/API boundaries/etc. xvi. If the code relies on external resources (e.g. another service), does it properly handle if the external resource is: a. Not available b. Timed out c. Appears to accept the request, but does not return a response d. Returns an unexpected response xvii. Is all user input validated before use? (Where user means any input coming from outside of the code, not just a physical person.) 10. Was any technical debt added? i. If so, why? It should be a really, really good reason why it was not addressed ii. It should be clearly documented where the tech debt was added. iii. A new Jira issue should be created detailing the tech debt and specifically what it will take to remove that debt. 11. Are there any TODO, FIXME or similar comments? If so, why? TODO is just a form of tech debt, so see above. 12. Was the Pull Request focused? It is best if the Pull Request is focused to one new feature to make it easier to review. 13. How many commits are involved in the Pull Request? If there is just one commit for a large number of unrelated changes this may point to lack of focus for the Pull Request. If there are a large number of commits, seriously consider rebasing down to a reasonable number. Strive for one commit per logical unit of work, but a feature may include multiple logical units of work. A logical unit of work is the smallest amount of work that can be completed and still have a functional project. i. For example, if I need to do a refactor in order to implement a new feature, it would make sense to have one commit for the refactor (and still have a functioning project), one commit for just the new feature after the refactor, and one overall pull request for both commits. 14. If there were UX changes, was a design review completed and approved? 15. Was any pertinent documentation

(internal and/or external) properly updated to reflect the code changes? 16. Is all PII/PHI communication and storage properly encrypted? 17. Was there any security review? Specific Checklists Here are some items to consider that are specific to a particular language and/or framework: Service Specific Checklist 1. If there were API or data model changes, was the documentation updated? 2. If there was a new API added: i. Does it require TLS/SSL? ii. Does it properly check the request for valid authentication and authorization? Does it properly return an error, if either fail? 3. Does it belong in that service? 4. Does the code use local resources (e.g. disk)? If so, will it still work if the next request goes to a different service? 5. Is the code scalable? Was it tested with more than one node? 6. Do new database queries correctly use existing indexes? Do they require new indexes? What is the performance impact if a query is used or not? 7. Do environment specific settings get loaded properly via the standard configuration mechanism for the service? Golang Specific Checklist 1. All external dependencies must be vendored using the standard Tidepool Golang dependency management tool (currently go dep). 2. Best practices for Golang use "early exit on error". Are all errors checked and does the error case return immediately while the success case continues on with the remainder of the function? 3. Use interfaces wherever possible, but when creating a new struct, return a concrete type, not an interface. Node.js Specific Checklist 1. All dependencies must specify an exact version (i.e. 1.2.3). Note that using any other form (e.g. 1.2.x) could yield non-reproducible builds if the lock file is regenerated. 2. Was a new dependency added? If so, was the yarn.lock file updated to match? 3. If the yarn.lock file was updated, why? Were the dependency updates reviewed and tested? React Specific Checklist 1. Are React packages used/added by this change available in React Native? For both iOS and Android? iOS Specific Checklist 1. Use guards to reduce conditional logic in the main body of a function. 2. Force unwrapping optionals is usually a bad idea. Guards or optional chaining is often a better solution. 3. Prefer let over var. 4. Make sure observer registrations are paired with unregistrations 5. Watch for retain cycles. Use unowned or weak references to break them. 6. Review for concurrency issues. Data accessed by multiple threads should be protected by a lock 7. Delegate methods should have the delegating object included in the method signature. Conclusion Finally, please consider this document to be evergreen. If you think of additional checklist items that may be useful, please feel free to add them above into the appropriate list and submit as Pull Request. If you really want to go crazy with a peer review, check out the OWASP Code Review Guide. Great read on how to keep PR reviews supportive and constructive for both the reviewer and reviewee: Unlearning toxic behaviors in a code review culture.

Tidepool Development Process Note: in late 2018, Tidepool switched to a new Git branching model that is very similar to GitFlow. The primary differences are the inclusion of required peer reviews for any merge into the master or develop branch and all tags are applied on release and hotfix branches, not the master branch. These relatively minor differences from GitFlow are necessary to

accommodate the Tidepool GitHub branch protection requirements. This document describes the overall development process, including Git branching model, from start to finish. Each specific repository will likely use a slightly modified development process to accommodate language and/or framework requirements. These differences will be highlighted in the repository README.md. This document assumes you already have a good working knowledge of using Git for software development. In fact, this document does not include all of the Git commands you will need to use during the normal development process. In particular how to add files, remove files, and create commits. If you don't have this knowledge, please learn that first! For the remainder of this document, all examples assume that the repository name is widgets, the feature branch name is add-tiny-widget, the release version is v1.4.0, and the hotfix version is v1.4.1. Setup Repository If starting development on a brand new Tidepool repository, follow the instructions in the Tidepool GitHub Processes document. This will create the repository with the default Tidepool settings, create the develop branch, and configure both the master and develop branch with the required branch protection settings. Clone the repository to your local machine. For example: $ mkdir tidepool $ cd tidepool $ git clone git@github.com:tidepool-org/widgets.git Feature Development Developing a new feature requires following several well-defined steps to ensure an orderly development process. Create Feature Branch Create a new feature branch off the latest develop branch. The feature branch name should accurately, but briefly, describe the feature. Use only lowercase ASCII letters, numbers, and dashes (not underscores, periods or any other symbols) in the branch name. For example: $ git checkout develop $ git pull $ git checkout -b add-tiny-widget develop Develop Feature Develop the new feature. Add unit tests to fully test the new feature. Ensure all tests pass. Manually test the feature, as necessary. Ensure the new feature matches all done criteria from the related Jira issue. Perform the Tidepool Code Peer Review Checklist process on the completed code to ensure everything it as it should be. Use standard git operations to add files, remove files, and create commits. Commit the feature code into one or more commits. Each commit should preferably be a single completed unit of work so that, if pushed to origin independently, the Continuous Integration system would build successfully and all tests would pass. Each commit title and description should accurately and completely describe, at a high-level, the changes contained within the commit. If you prefer to use Git to commit code frequently, even if it is incomplete or does not work, perhaps for "backup" purposes or just because it is the end of the day, please consider rebasing your feature branch and squash code into one commit per completed and tested unit of work. Remember that once commits are merged back into the develop and master branches they are a permanent part of git history for the entire world to see. On the other hand, do not unnecessarily squash a set of unrelated work into a single commit. This makes a peer review considerably more difficult. So, strive for a balance of focused, but complete commits. For example, if you need to refactor code before you can implement a new feature, it would make sense to have one commit for the refactor (and still have a functioning project) and one commit for the new fea-

ture after the refactor. This will make the review easier and may help isolate whether a newly introduced bug is part of the refactor or the new feature. Note: If you ever rebase a feature branch that has already been pushed to origin, do not force push the branch (i.e. do not use -f or –force with git push). Instead, create a new feature branch with the same name but append a “dot number” to the end. For example, if the feature branch was add-tiny-widget then any rebased branches would follow the pattern add-tiny-widget.1, add-tiny-widget.2, add-tiny-widget.3, etc. Delete the previous branch both locally and remotely once you have pushed the new rebased branch. Continue working with the new rebased branch. Merge Latest Develop Branch into Feature Branch Once you are confident that the feature branch is complete and ready for review, perform a final merge of the develop branch into the feature branch, as it may have changed since you originally created the feature branch. For example: $ git checkout develop $ git pull $ git checkout add-tiny-widget $ git merge –no-ff develop Manually resolve any conflicts. Since there may have been automatic or manually resolved conflicts, review any changes and perform a final, full test of the code. All tests must pass. Push Feature Branch Now that you are certain the feature branch is complete and contains the latest from the develop branch, push the completed feature branch to origin, if you have not already done so. For example: $ git push -u origin add-tiny-widget Deploy Feature Branch (Optional) Rarely it will be necessary to deploy an in-progress feature branch. Typically this would be to test a specific condition only present when deployed or released. If this occurs, use a feature branch label or tag of the form v-., where is the version of the active release branch or, if no active release branch is present, the latest deployed version, and starts at 1 and allows for multiple labels or tags. The version should follow Semantic Versioning 2.0.0 rules. A feature branch label or tag of this form is still a valid Semantic Version. Note the v prefix. For example, v1.4.0-add-tiny-widget.1. Peer Review Feature Branch into Develop Branch Once you have pushed the completed feature branch to origin and verified that any CI build completes successfully, create a pull request for the peer review. Use either the GitHub Web interface or the command line to create the pull request. For example: $ open https://github.com/tidepool-org/widgets/pull/new/develop…add-tiny-widget Ensure the base branch is develop and the compare branch is your final feature branch, particularly if you have rebased one or more times. Update the title and description, if desired. It may be helpful to add extra notes for the peer reviewer. In the upper right corner, add one or more Reviewers for this pull request. We no longer specify reviewers via @mentions in the description, but instead use the GitHub Reviewers mechanism. Generally, only one reviewer is required per pull request. However, if you desire multiple reviewers, then all reviewers must review and approve the pull request. If you are specified as a reviewer, then you must review and approve the pull request even if there are other reviewers. It is not “one of”, it is “all”. We want to prevent the situation where the reviewers all think someone else is doing the review. If you aren’t sure who should review the pull request, then determine that outside of GitHub (e.g. via Slack) before creating the pull request. We want it to be abundantly clear who is responsible for reviewer the

pull request. For example:

Note the pull request reviewer is set to pazaan in the upper right corner. Click Create pull request. You may wish to notify the reviewer of the pull request via Slack. Add the pull request to the Jira issue by using a Smart Commit tag. Jira will automatically pick it up. If you are a requested reviewer, but you will not be able to perform the review in a timely manner, please reach out to the requestor and explain the situation. The requestor may be fine waiting or may decide to choose another reviewer. The reviewer should use the Tidepool Code Peer Review Checklist to help perform the peer review. Coordinate with the reviewer to discuss any questions, comments or suggestions. Strive to keep all questions, comments, suggestions, and final decisions in the pull request comment stream so the entire peer review is contained within the pull request itself for future reference. It is acceptable to have in-depth side conversations outside of the pull request (e.g. Slack, email, video chat), but, at a minimum, capture the initial issue, final decision, and its rationale in the pull request comments. If desired or necessary, update your feature branch and commit any changes. Once complete, follow Push Feature Branch instructions. You may also need to follow Merge Latest Develop Branch into Feature Branch instructions before pushing if the develop branch has recently changed. Once you and the reviewer are satisfied with the results, the reviewer should approve your pull request and attach the completed Tidepool Code Peer Review Checklist. For example:

Merge Feature Branch into Develop Branch Once the pull request is approved, the finished feature branch may be merged into the develop branch to definitely add it to the upcoming release. If the develop branch has changed since you last merged then follow Merge Latest Develop Branch into Feature Branch, Push Feature Branch, and Peer Review Feature Branch instructions again. (The reviewer does not need to complete a full review, but does need to review and approve the final merged code for any issues with possible merge conflicts. This is enforced by the required GitHub branch protection settings.) Merge the feature branch into the develop branch. For example: $ git checkout develop $ git pull $ git merge –no-ff add-tiny-widget $ git push origin develop The –no-ff flag forces the merge to always create a new commit, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of the feature branch. Delete Feature Branch Assuming the merge and push were successful, delete the feature branch. For example: $ git branch -d add-tiny-widget $ git push origin :add-tiny-widget Release Process Create Release Branch from Develop Branch Once all features to be included in a release are merged into the develop branch, create a release branch from the develop branch. Release branches are named with the release- prefix and the version number of the release. The version number here does not include the v prefix. The version should follow Semantic Versioning 2.0.0 rules. However, normally releases will always use a patch version of 0, with the patch version being reserved for a hotfix. For example, if the previous version was v1.3.1 and the develop branch is “backwards-compatible” with v1.3.1, then the release branch would be named release-1.4.0. $ git checkout develop $ git pull $ git checkout -b

release-1.4.0 Perform any repository-specific steps to version and tag the release (e.g. update package.json and create/push the GitHub tag) with a pre-release label. Since this release must be tested and approved by QA and may require one or more changes before it can be deployed, do not initially label or tag the release with its final version number. Instead, use a pre-release label or tag of the form v-release. where starts at 1 and allows for multiple labels or tags. A pre-release label or tag of this form is still a valid Semantic Version. Note the v prefix. For example, v1.4.0-release.1. Push Release Branch Push the release branch and any tags to origin. For example: $ git push -u –tags origin release-1.4.0 Deploy and Test Release Branch Wait for the official, tagged build artifacts to be available. Deploy the release to a test environment and have QA perform all necessary tests. Patch Release Branch If QA finds one or more issues that must be addressed before the release can be deployed, update the code and add commits to the release branch. Increment the for every subsequent pre-release label. For example, v1.4.0-release.2. Follow Push Release Branch and Deploy and Test Release Branch instructions after every patch. Finalize Release Once QA officially approves the release and it is approved for deployment, the release should be finalized. Perform any repository-specific steps to version and tag the release (e.g. update package.json and create/push the GitHub tag) with its final release label. For example, v1.4.0. Peer Review Release Branch into Master Branch Due to the Tidepool GitHub branch protection settings, it is necessary to perform a pull request to approve merging the release branch into the master branch. Use either the GitHub Web interface or the command line to create the pull request. For example: $ open https://github.com/tidepool-org/platform/pull/new/master…release-1.4.0 Ensure the base branch, compare branch, title, description, and reviewers are correct. The description should note that the pull request is simply to review the merge from the release branch back into the master branch. Merge Release Branch into Master Branch Once the pull request is approved, merge the release branch into the master branch. For example: $ git checkout master $ git pull $ git merge –no-ff release-1.4.0 $ git push origin master Peer Review Release Branch into Develop Branch Due to the Tidepool GitHub branch protection settings, it is necessary to perform a pull request to approve merging the release branch into the develop branch. Use either the GitHub Web interface or the command line to create the pull request. For example: $ open https://github.com/tidepool-org/platform/pull/new/develop…release-1.4.0 Ensure the base branch, compare branch, title, description, and reviewers are correct. The description should note that the pull request is simply to review the merge from the release branch back into the develop branch. Merge Release Branch into Develop Branch Once the pull request is approved, merge the release branch into the develop branch. For example: $ git checkout develop $ git pull $ git merge –no-ff release-1.4.0 $ git push origin develop Delete Release Branch Assuming the previous steps were successful, delete the release branch. $ git branch -d release-1.4.0 $ git push origin :release-1.4.0 Hotfix Process Sometimes there is an urgent need to fix a critical issue with production. Since it is not advisable to interrupt normal feature development and release process, the hotfix process should be followed. Cre-

ate Hotfix Branch from Master Branch Create a hotfix branch from the master branch. Hotfix branches are named with the hotfix- prefix and the version number of the hotfix. The version number here does not include the v prefix. The version should follow Semantic Versioning 2.0.0 rules. Normally, hotfixes will only increment the patch version as this allows any release branches currently in progress to retain their version number. For example, if the previous version was v1.4.0 then this hotfix branch would be named hotfix-1.4.1. $ git checkout master $ git pull $ git checkout -b hotfix-1.4.1 Perform any repository-specific steps to version and tag the hotfix (e.g. update package.json and create/push the GitHub tag) with a pre-hotfix label. Since this hotfix must be tested and approved by QA and may require one or more changes before it can be deployed, do not initially label or tag the hotfix with its final version number. Instead use a pre-hotfix label or tag of the form v-hotfix. where starts at 1 and allows for multiple labels or tags. A pre-hotfix label or tag of this form is still a valid Semantic Version. Note the v prefix. For example, v1.4.1-hotfix.1. Push Hotfix Branch Push the hotfix branch and any tags to origin. For example: $ git push -u –tags origin hotfix-1.4.1 Deploy and Test Hotfix Branch Wait for the official, tagged build artifacts to be available. Deploy the hotfix to a test environment and have QA perform all necessary tests. Patch Hotfix Branch If QA finds one or more issues that must be addressed before the hotfix can be deployed, update the code and add commits to the hotfix branch. Increment the for every subsequent pre-hotfix label. For example, v1.4.1-hotfix.2. Follow Push Hotfix Branch and Deploy and Test Hotfix Branch instructions after every patch. Finalize Hotfix Once QA officially approves the hotfix and it is approved for deployment, the hotfix should be finalized. Perform any repository-specific steps to version and tag the hotfix (e.g. update package.json and create/push the GitHub tag) with its final hotfix label. For example, v1.4.1. Peer Review Hotfix Branch into Master Branch Due to the Tidepool GitHub branch protection settings, it is necessary to perform a pull request to approve merging the hotfix branch into the master branch. Use either the GitHub Web interface or the command line to create the pull request. For example: $ open https://github.com/tidepool-org/platform/pull/new/master…hotfix-1.4.1 Ensure the base branch, compare branch, title, description, and reviewers are correct. The description should note that the pull request is simply to review the merge from the hotfix branch back into the master branch. Merge Hotfix Branch into Master Branch Once the pull request is approved, merge the hotfix branch into the master branch. For example: $ git checkout master $ git pull $ git merge –no-ff hotfix-1.4.1 $ git push origin master “Target” Release or Develop Branch If there is active release branch, then use that release branch as the “target” branch in the following few sections. If there is not an active release branch, then use the develop branch as the “target” branch. Merge “Target” Branch into Hotfix Branch Note: See “Target” Release or Develop Branch before continuing. Since the “target” branch may have changes that are not yet included in the hotfix branch, it is necessary to merge those into the hotfix branch before it can be merged back into the “target” branch. This MUST be completed AFTER the hotfix branch is merged into the master branch. For example: $ git checkout release-1.5.0 $ git pull $

git checkout hotfix-1.4.1 $ git merge –no-ff release-1.5.0 You may have to manually resolve any conflicts. Since there may have been automatic or manually resolved conflicts, review any changes and perform a final, full test of the code. All tests must pass. Push Hotfix Branch Note: See “Target” Release or Develop Branch before continuing. If there were changes in the “target” branch merged into the hotfix branch then push the hotfix branch to origin. For example: $ git push -u origin hotfix-1.4.1 Peer Review Hotfix Branch into “Target” Branch Note: See “Target” Release or Develop Branch before continuing. Due to the Tidepool GitHub branch protection settings, it is necessary to perform a pull request to approve merging the hotfix branch into the “target” branch. Use either the GitHub Web interface or the command line to create the pull request. For example: $ open https://github.com/tidepool-org/platform/pull/new/release-1.5.0…hotfix-1.4.1 Ensure the base branch, compare branch, title, description, and reviewers are correct. The description should note that the pull request is simply to review the merge from the hotfix branch back into the “target” branch. Merge Hotfix Branch into “Target” Branch Note: See “Target” Release or Develop Branch before continuing. Once the pull request is approved, merge the hotfix branch into the “target” branch. For example: $ git checkout release-1.5.0 $ git pull $ git merge –no-ff hotfix-1.4.1 $ git push origin release-1.5.0 Delete Hotfix Branch Assuming the previous steps were successful, delete the hotfix branch. $ git branch -d hotfix-1.4.1 $ git push origin :hotfix-1.4.1

Tidepool External Code Dependency Considerations If you are thinking about adding a new external code dependency to a Tidepool project, please give serious thought to the following considerations before committing to the dependency. There are few hard-and-fast rules, except perhaps the license requirement, so use your best judgement while taking these considerations into account. When in doubt, pull in another engineer or engineering manager to help with the decision. This document applies to code. If that code depends on a service API for its operation (e.g. it is a vendor-provided SDK), you may also want to keep in mind the related considerations for service APIs. Those are covered in Tidepool External Service API Dependency Considerations. You should repeat this process for any additional sub-dependencies that a dependency may pull in. Common Considerations 1. What license does the dependency use? The MIT, BSD, and Apache licenses are preferred and may be required depending on how you will be using the dependency. If there is any question, get approval from Howard. 2. Are there any reasonable alternatives? Spend a few minutes searching for alternatives. If there are alternatives, then use this process with those, too, as a comparison. 3. How much code is actually going to be used from the dependency? If only a small amount of code compared to the overall size of the dependency, then consider finding an alternative or just implementing it locally. Even consider copy/paste if the license allows for it, but remember to give credit to the original work. 4. What sub-dependencies does that dependency have? How much extra, unused stuff is coming along for the ride? Apply this process to those sub-dependencies. 5. What sort of security record does the code or project have? Do they have any outstand-

ing security reports? What is the impact of previously reported security and other types of bugs? i. https://cve.mitre.org/cve/search_cve_list.html ii. https://www.sourceclear.com/vulnerability-database/search iii. https://www.openhub.net/ 6. How active is the community for the dependency? i. How many active contributors? Recent turnover? (See event-stream incident) ii. What is the frequency and recentness of commits? iii. How many open issues are there? What is the impact of these open issues? Do they represent significant problems or just minor problems and feature requests? iv. How frequently are issues closed? What is the responsiveness? Why were they closed? Were they actually fixed or were they just closed for lack of progress? v. Review GitHub Insights. 7. How much interest is there from the outside community (e.g. stars, watches, forks)? i. Is there any fork that may be more appropriate? Some forks are more active, include additional bug fixes, and are an overall better choice, particularly if work on the primary project has stalled. Repeat this process for any forks that might be viable. 8. Review the code. i. Consider applying all or part of the Tidepool Code Peer Review Checklist against the dependency. ii. What does the code look like (readability, understandability, etc.)? iii. Does the code conform to any coding standards or best practices for the language? iv. What documentation is available? a. For external use? b. For internal development and maintenance? v. What test coverage does the dependency have? a. Run the tests. Do the tests pass? b. Are they executed automatically on each PR/commit? vi. What update strategy does the dependency use for backwards compatibility? Is anything documented? If not, review the change log or actual code changes between previous versions. vii. Does it come with build instructions so someone else can build this from scratch, if needed? 9. How will we "lock" a particular reviewed version of the dependency (and any sub-dependencies) to ensure we get an exact reproducible build (even years later)? Can/should the dependencies be vendored? 10. Consider the impact of the dependency to our HIPAA compliance requirements Specific Considerations Here are some items to consider that are specific to a particular language and/or framework: Golang Specific Considerations 1. What version(s) of Golang has the dependency been built and tested against? If it does not explicitly support the latest version the Tidepool repository uses, double check the Golang release notes for that version to see if are any potential conflicts. 2. Does the dependency use the standard Golang formatting and linting tools? 3. All external dependencies must be vendored using the standard Tidepool Golang dependency management tool. The reasons for this are: i. The go tool chain does not guarantee that a dependency will be available in the future. For example, a developer can delete a repo. Without a store of all published versions, the dependency would disappear. ii. The dependencies must be available at build time. Vendoring simply stores those dependencies in the local repo. At present this consists of over 500K lines of Go code. Since space is basically free, this is not a big concern. Node.js Specific Considerations 1. Are you using a yarn.lock or a package-lock.json for you repo to ensure that sub-dependencies are locked down? 2. Only use fixed versions of dependencies, i.e., don't use version ranges (^,~) React Specific Considerations None iOS

Specific Considerations None Conclusion Finally, please consider this document to be evergreen. If you think of additional considerations that may be useful, please feel free to add them above into the appropriate list and submit a Pull Request.

Tidepool External Service API Dependency Considerations If you are thinking about adding a new external service API dependency to a Tidepool project, please give serious thought to the following considerations before committing to the dependency. There are few hard-and-fast rules, except perhaps the security and privacy requirements, so use your best judgement while taking these considerations into account. When in doubt, pull in another engineer or engineering manager to help with the decision. This document applies mainly to the service itself, and the APIs it exposes for our use. If that service API is accessed through vendor-provided SDK, you may also want to keep in mind the related considerations for external dependencies as code. Those are covered in Tidepool External Code Dependency Considerations. You should repeat this process for any additional sub-dependencies that a service API introduces. In other words, other services that are used through calls to service API. Common Considerations 1. Are there reasonable alternatives? Including in-house? (build vs. buy/license) 2. Review the SLA i. What is their uptime promise? ii. What is their escalation process? iii. Is it supported 24/7 globally? iv. Do they have an uptime dashboard? v. What is their EOL policy for legacy APIs? 3. Is the API available in all the geographies we need? i. Is the performance adequate from other geographies? ii. What is their DR plan? iii. Do they have automatic failover to another geography? iv. If the service stores data persistently, is it subject to GDPR and similar laws? 4. How will it affect us if the service suffers a prolonged downtime? This could be outside of their control (AWS us-east-1 outage) 5. Is the API available for different development environments? (dev, stg, prod, …). Are they wholly separate or mixed (e.g. user/account IDs)? Are the non-production environments sufficiently representative of performance etc. of the production environment. Do you have a clear understanding of how they differ? 6. Is the API rate limited? Does that rate meet our current and forecast needs? 7. Does the API support bulk operations? Are they needed by our use-cases? 8. Is there an SDK available in the language(s) we use? Do we need to write our own? Is the wire protocol tolerable (JSON, XML, …)? 9. Is their code open source? Can you look at it and use the same considerations as Tidepool External Code Dependency Considerations? Is it possible to install it locally and we manage it? Does that make sense? 10. What plan/consideration should be made if it were to go away overnight? How likely is that? 11. What does their funding, business model, expected longevity look like? Security & Privacy Specific Considerations 1. How is authentication and authorization handled? Whole service vs. per-user? 2. Is all traffic secured with TLS/SSL? Are their certificates up-to-date? 3. Will it be handling any PII/PHI data in transit or at rest? Do they comply with HIPAA requirements? Will they sign a BAA, if necessary? If in doubt, consult Howard and Brandon. 4. What sort of security record does the code or project have? Do they have any out-

standing security reports? What is the impact of previously reported security and other types of bugs? i. https://cve.mitre.org/cve/search_cve_list.html ii. https://www.sourceclear.com/vulnerability-database/search iii. https://www.openhub.net/ 5. Do we plan to store any Tidepool or customer data there long-term? i. Data encrypted at rest? ii. What is the plan for purging (account deletion, GDPR purge requests, …) Conclusion Finally, please consider this document to be evergreen. If you think of additional considerations that may be useful, please feel free to add them above into the appropriate list and submit a Pull Request.

Tidepool GitHub Processes Create Repository Follow these directions to create a new Tidepool GitHub repository that conforms to the latest Tidepool standards for repository name, license, default branch, and branch protection. Prepare Choose a descriptive name for the repository. It must be unique among all other Tidepool GitHub repositories. It must not include tidepool in the repository name, that is implied in the full URL (e.g. github.com/tidepool-org/development). Prefer lowercase and dashes to uppercase and underscore for consistency. Code names are no longer allowed. If you have questions, ask one of the Tidepool GitHub administrators. Decide whether the repository should have public or private visibility. The repository should be private only if it will contain secrets, security-related matters, third-party proprietary information, or anything else that must not be made public. All other repositories should be public. If you are not sure, ask Howard or Tapani. Login to GitHub as an Tidepool GitHub administrator. If you aren't a Tidepool GitHub administrator, then you'll need to get one to do the rest. Create Repository Browse to https://github.com/organizations/tidepool-org/repositories/new and: 1. Ensure the Owner is set to tidepool-org. 2. Enter the Repository name, as chosen above. 3. Add a Description, if desired, but this can easily be changed later. 4. Select Public or Private visibility, as chosen above. 5. Select Initialize this repository with a README. 6. From the Add .gitignore popup menu choose None. 7. From the Add a license popup menu choose BSD 2-Clause "Simplified" License. For example:

Click Create repository. Create Develop Branch Once the repository is created, create the develop branch so the repository can follow the Tidepool Development Process, which closely models GitFlow. On the main repository page, click the Branch popup menu, type develop, and click Create branch: 'develop' from 'master'. For example:

Configure Repository Settings Follow Default Repository Settings instructions below. Default Repository Settings Configure Options Click the Settings icon in the upper-right corner. In the left menu, click Options, and: 1. Deselect Wikis, Issues, and Projects, unless otherwise required. 2. Deselect Allow squash merging and Allow rebase merging For example:

Configure Teams Click the Settings icon in the upper-right corner. In the left menu, click Collaborators & teams and: 1. Click Add a team, choose Administrators and set permission to Admin 2. Click Add a team, choose Employees

and set permission to Read 3. Click Add a team, choose the appropriate group (e.g. Engineering, Analytics) and set permission to Write For example:

Configure Default Branch and Branch Protection Click the Settings icon in the upper-right corner. In the left menu, click Branches. Select develop as the Default branch and click Update. Click I understand, update the default branch. For both the master and develop branches, click Add rule to add a Branch protection rule, and: 1. Enter the branch name (i.e. master or develop) 2. Select Require pull request reviews before merging 3. Select Dismiss stale pull request approvals when new commits are pushed 4. Select Require status checks to pass before merging 5. Select Require branches to be up to date before merging 6. Note: If and when you add a Continuous Integration or other connection to this repository, you'll need to come back here and enable all available status checks 7. Select Include administrators For example:

Click Create. Repeat for both master and develop branches. For example:

13.16 Purpose

This document establishes a standard method for completing, identifying, collecting, filing, storing, and dispositioning quality records at Medical Data Networks, LLC (MDN). Quality records are maintained to provide supporting evidence of the conformity, implementation, and effective operation of the QMS.

13.17 References

1. 21 CFR 820
2. FDA
3. Quality System Regulation
4. ISO 13485:2016 Clause 4.2.5

13.18 Responsibilities

1. The CEO and VP-level employees are responsible for overseeing and maintaining this standard operating procedure and for assuring that all employees are trained in its requirements.
2.

13.19 It is the responsibility of all employees, contractors and departments at Medical Data Networks to adhere to this procedure.

repository: "github.com/ehwest/mdn_qms" folder: "PDP_Product_Development_Process" title: "PDP_phase_4_Release_to_Production.md" authors:

- github.com/ehwest approvers:
- github.com/bewest approval_date: "2020-10-01" —

13.20 Purpose

This document establishes a standard method for completing, identifying, collecting, filing, storing, and dispositioning quality records at Medical Data Networks, LLC (MDN). Quality records are maintained to provide supporting evidence of the conformity, implementation, and effective operation of the QMS.

13.21 References

1. 21 CFR 820
2. FDA
3. Quality System Regulation
4. ISO 13485:2016 Clause 4.2.5

13.22 Responsibilities

1. The CEO and VP-level employees are responsible for overseeing and maintaining this standard operating procedure and for assuring that all employees are trained in its requirements.
2.

13.23 It is responsibility of all employees, contractors and departments at Medical Data Networks to adhere to this procedure.

repository: "github.com/ehwest/mdn_qms" folder: "PDP_Product_Development_Process" title: "PDP_phase_5_Post_Market_Surveillance" authors:

- github.com/ehwest approvers:
- github.com/bewest approval_date: "2020-10-01" —

13.24 Purpose

This document establishes a standard method for completing, identifying, collecting, filing, storing, and dispositioning quality records at Medical Data Networks, LLC (MDN). Quality records are maintained to provide supporting evidence of the conformity, implementation, and effective operation of the QMS.

13.25 References

1. 21 CFR 820
2. FDA
3. Quality System Regulation
4. ISO 13485:2016 Clause 4.2.5

13.26 Responsibilities

1. The CEO and VP-level employees are responsible for overseeing and maintaining this standard operating procedure and for assuring that all employees are trained in its requirements.
2. It is the responsibility of all employees, contractors and departments at Medical Data Networks to adhere to this procedure.